Die approbierte Originalversion dieser Dissertation ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

Dissertation

Component Based Communication Middleware for AUTOSAR

Ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter der Leitung von

o.Univ.Prof. Dr.techn. Mehdi Jazayeri
und
Dr.techn. Karl M. Göschka

184-1
Institut für Informationssysteme
Arbeitsbereich für Verteilte Systeme

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

Dipl.-Ing.(FH) Dietmar Schreiner
Matr.Nr. 9026735

Wien, Oktober 2009 ______________________________

This work has been partially funded by the FIT-IT Embedded Systems initiative of the Austrian Federal Ministry of Transport, Innovation, and Technology, and managed by Eutema and the Austrian Research Agency FFG within research project COMPASS under contract 809444, and by the 7th EU R&D Framework Program within research project ALL-TIMES under contract 215068.

Kurzfassung

Aufgrund ständig wachsender Anforderungen und innovativer Neuerungen im Automobilbereich haben sich automotive Elektroniksysteme innerhalb der letzten Jahre zu komplexen, verteilten Echt-Zeit-Systemen entwickelt. Um der daraus resultierenden Kostenschere zwischen steigender Komplexität und vertretbarem Entwicklungsaufwand entgegenzuwirken, legt der junge Industriestandard *AUTOSAR Komponentenbasierte Software-Entwicklung* (CBSE) als Methodik für zukünftige automotive Anwendungen fest.
Durch den Einsatz des Komponentenparadigmas auf Applikationsebene wird eine klare Aufgabentrennung innerhalb automotiver Software festgelegt: Anwendungen werden aus wiederverwendbaren und austauschbaren Bausteinen, den Software-Komponenten, zusammengesetzt, deren Implementierung nur anwendungsspezifische Funktionalität enthält. Infrastrukturelle Funktionalität wird durch standardisierte Komponenten-Middleware, der *AUTOSAR Basic Software* und dem *AUTOSAR Run-Time Environment*, realisiert, und den Software-Komponenten zur Verfügung gestellt. Die so gewählte Architektur soll zu einer Steigerung der Wiederverwendbarkeit, Wartbarkeit und Qualität automotiver Software, und daher zu einer Eindämmung von Kosten und Entwicklungszeit führen.
Die vom *AUTOSAR* Standard definierte Middleware ist allerdings als Schichtenarchitektur spezifiziert, die nur im Groben an Anwendungsanforderungen angepasst werden kann. Häufig ist daher ungenutzte Funktionalität in eingesetzter Middleware enthalten, und trägt dazu bei, daß herkömmliche *AUTOSAR* konforme Komponenten-Middleware oft überdimensioniert und verschwenderisch im Umgang mit verfügbaren Ressourcen ist. Diese Tatsache stellt gerade im Bereich von ressourcenbeschränkten automotiven eingebetteten Systemen eine Schwäche des *AUTOSAR* Standards dar.
Die vorliegende Dissertation trägt zur Entschärfung der Middleware-Problematik in *AUTOSAR* bei, indem sie den Einsatzbereich von CBSE über die Anwendungsschicht hinaus auf die Komponenten-Middleware, im speziellen auf das Kommunikationssubsystem, ausdehnt.
Dazu wird in einem ersten Schritt gezeigt, wie die schichtenbasierte Architektur der *AUTOSAR* Middleware durch eine komponentenbasierte ersetzt werden kann. Die vorgeschlagene komponentenbasierte Architektur wird mittels einer statischen Analyse, der *Kohäsionsanalyse*, aus einer herkömmlichen industriellen Implementierung extrahiert, und bietet bei größerer Flexibilität vollwertigen, standardkonformen Ersatz für diese. Zusätzlich werden auf unterschiedliche Anwendungsfälle hin optimierte Implementierungsvarianten der errechneten Middleware-Komponentenklassen definiert, durch deren Ein-

satz maßgeschneiderte, komponentenbasierte Middleware erzeugt werden kann.
In einem zweiten Schritt wird eine Modell-Transformation, die *Konnektor-Transformation*, für *AUTOSAR* Applikationsmodelle vorgestellt, die eine automatische Synthese von applikationsspezifisch optimierter Middleware bewerkstelligt. Die *Konnektor-Transformation* ersetzt alle Vorkommnisse von so genannten expliziten Konnektoren in plattformunabhängigen Applikationsmodellen durch Middleware-Komponentenarchitekturen, bestehend aus vorgefertigten Middleware-Komponenten. Die so generierten und dem Verteilungsszenario der Applikation entsprechenden plattformspezifischen Modelle enthalten als Ergebnis die Applikation, sowie genau jene Middleware-Funktionalität, die von der Applikation benötigt wird. Schlussendlich werden die plattformspezifischen Modelle in ausführbaren Code für jeden Systemknoten der (eventuell verteilten) Applikation transformiert.
Um die entwickelte Methodik zu evaluieren, wurde eine einfache automotive Anwendung sowohl in herkömmlicher als auch in der vorgeschlagenen Art und Weise entwickelt. Wie Messungen zeigen, weist die synthetisierte Kommunikationsmiddleware eine eindeutige Verbesserung im Bezug auf Programmgröße und Prozessorlast auf: Synthetisierte Middleware ist bezüglich ihres Programmspeicherbedarfs um bis zu 30% kleiner, und benötigt um bis zu 10% weniger Ausführungszeit, als herkömmliche *AUTOSAR* Kommunikationsmiddleware.

Abstract

Driven by steadily increasing requirements of innovative applications, automotive electronics has evolved into highly dependable, distributed, real-time embedded systems. To close the rising gap between increasing complexity and affordable costs, the upcoming automotive software standard *AUTOSAR* constitutes *Component Based Software Engineering* (*CBSE*) as development methodology for future automotive applications. *CBSE* introduces a clear separation of concerns into *AUTOSAR*'s system architecture: Any application is built from reusable and exchangeable so called *Software Components* that deal with business logic only, whereas standardized infrastructural services are provided by component middleware—the *AUTOSAR Basic Software* and the *AUTOSAR Run-Time Environment*. This design leads to an increase in reusability, maintainability, and application quality, and hence to a reduction of costs and time-to-market. However, *AUTOSAR* component middleware is specified as layered software architecture that is customizable on a coarse-grained level only, and thus tends to be heavy-weight and impractical in resource constrained embedded systems.
This thesis contributes by extending the scope of *CBSE* within *AUTOSAR* beyond the application layer to the component middleware, especially to its communication subsystem. In a first step, the layered *AUTOSAR* middleware architecture is replaced by a component based design that is extracted from an existing layered reference implementation by static analysis—the *Cohesion Analysis*. The proposed component based communication middleware completely resembles the standard's functionality, but is more flexible in terms of application specific customization. In addition, use-case based variants of identified component classes are defined to enable improved middleware optimization. In a second step, a model transformation for *AUTOSAR* application models is specified that enables automatic middleware synthesis in line with the *AUTOSAR* methodology. The *Connector Transformation* injects middleware component architectures in place of all explicit connectors within the application models. Thereby, platform-specific models for each system node are generated, which completely and solely reflect the application's middleware requirements.
To evaluate the proposed approach, it is applied to a simple automotive application. The hereby gained synthesized communication middleware shows an improvement of nearly 30% with respect to its memory footprint and a reduction in CPU usage of up to 10% compared to its conventional counterpart.

Acknowledgements

I would like to thank all those persons, who supported me during the process of writing this thesis: Dr. Karl M. Göschka for supporting and guiding my research activities, o.Univ.Prof. Dr. Mehdi Jazayeri for being my senior adviser and providing helpful pointers, Prof. Dr. Uwe Aßmann for reviewing the thesis, o.Univ.Prof. Dr. Jens Knoop and Dr. Markus Schordan for fruitful discussions on static analysis, Dr. Martin Horauer for his support within project COMPASS, Dr. Thomas Galla for the numerous *AUTOSAR* specific discussions, and finally Dipl.-Ing.(FH) Wolfgang Forster and Dipl.-Ing. Gergö Barany for their support at the proof-of-concept implementation and the measurements.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Approach . . . 4
1.3 Contribution . . . 5
1.4 Thesis Structure . . . 7

2 Technological Baseline and State-of-the-Art **9**
2.1 Component Based Software Engineering . . . 9
2.1.1 Components . . . 10
2.1.1.1 Component Definition . . . 11
2.1.1.2 Interfaces . . . 13
2.1.1.3 Ports . . . 14
2.1.2 Connectors . . . 15
2.1.3 Component Middleware . . . 17
2.2 Model Driven Development . . . 20
2.2.1 Viewpoints and Models . . . 21
2.2.2 Transformations . . . 23
2.2.3 MDA Development Phases . . . 25
2.3 Summary . . . 27

3 Automotive Open System Architecture **28**
3.1 History of AUTOSAR . . . 28
3.2 Hardware Architecture . . . 29
3.3 Software Architecture . . . 30
3.3.1 Application Software Architecture . . . 31
3.3.2 System Software Architecture . . . 32
3.4 AUTOSAR Software Components . . . 35
3.4.1 Software Component Definition . . . 36
3.4.2 Composition and Interaction . . . 38
3.4.3 Component Middleware . . . 40
3.5 Methodology . . . 41

3.6 Summary 42

4 COMPASS Middleware 43
4.1 Component Based Middleware 43
4.2 Component Model 45
4.2.1 Data Types 45
4.2.2 Component Definition 47
4.2.2.1 Embodiment of COMPASS Components . . . 48
4.2.2.2 Component Type 51
4.2.3 Composition and Interaction 53
4.3 Middleware Components 54
4.3.1 Manual Component Classification 54
4.3.2 Component Recognition by Static Analysis 56
4.3.2.1 Basic Principle 57
4.3.2.2 Cohesion Analysis Algorithm 59
4.4 Summary 65

5 Middleware Synthesis 66
5.1 Methodology 66
5.2 COMPASS Application Models 69
5.2.1 Explicit Connectors 69
5.2.1.1 Deployment Anomaly 71
5.2.1.2 Interaction Styles 73
5.2.2 Contracts 74
5.2.2.1 Emerging Contracts 76
5.3 Connector Transformation 77
5.3.1 Connector Templates 79
5.3.1.1 Sender-Receiver Connector Architecture . . . 80
5.3.1.2 Client-Server Connector Architecture 82
5.3.2 Connector Transformation Algorithm 84
5.3.3 Model-to-Code Transformation 89
5.4 Summary 90

6 Discussion 92
6.1 Proof of Concept 92
6.1.1 Middleware Component Recognition 92
6.1.1.1 Comparison of Approaches 94
6.1.2 Testbed Application 96
6.1.3 Middleware Synthesis 98
6.2 Related Work 102
6.2.1 Component Models 102

6.2.2 Static Analysis 104
6.2.3 Model Driven Development 105
6.3 Future Work 106
6.4 Conclusion 107

Appendices **109**

A SPEM **109**

B Used UML 2 Diagram Types **112**

C FlexRay Communication Layers **114**

D Manual Annotations for Component Recognition **117**
D.1 Functions 117
D.2 Structure Fields 120

List of Figures **121**

List of Algorithms **123**

List of Tables **124**

Bibliography **125**

Publications **135**

Curriculum Vitae **137**

Abbreviations

AST .... Abstract Syntax Tree
AUTOSAR ...... Automotive Open System Architecture
BSW .... Basic Software
CBSE .... Component Based Software Engineering
CC .... Communication Controller
CIM .... Computation Independent Model
COMPASS .... Component Based Automotive System Software
ECU .... Electronic Control Unit
IPC .... Intra Process Connector
ISR .... Interrupt Service Routine
MCU .... Micro Controller Unit
MDA .... Model Driven Architecture
MOF .... Meta Object Facility
MWBB .... Middleware Building Block
OEM .... Original Equipment Manufacturer
OMG .... Object Management Group
PDU .... Protocol Data Unit
PIM .... Platform Independent Model
PSM .... Platform Specific Model
QoS .... Quality-of-Service
QVT .... Query/View/Transformation
RAM .... Random Access Memory
ROM .... Read-Only Memory
RTE .... Run-Time Environment
SPEM .... Software Process Engineering Metamodel
UML .... Unified Modeling Language
VFB .... Virtual Function Bus
XML .... Extensible Markup Language
XNC .... Extra Node Connector
XPC .... Extra Process Connector

Chapter 1

Introduction

1.1 Motivation

Driven by market demands, the application of automotive embedded systems experienced a significant upturn over the last few years. A wide variety of new fields of application introduced new market opportunities, but also new challenges to the systems' developers. Functionality like steer/break-by-wire, ambient-intelligence or multimedia functionality is exemplary for those newly arising services, modern vehicles provide. State-of-the-art vehicles contain more than 70 Electronic Control Units (ECUs), typically connected by up to 10 diverse bus systems [Han05]. Today's automotive applications are no longer simple programs executed on one single ECU. In fact, they are heterogeneous software systems in distributed and often safety and mission critical environments.

By considering the large number of automobiles manufactured every year—in 2007 a total of 73, 2 million vehicles [OIC08] have been produced worldwide—and the rather long life-cycles of more than 10 years, it becomes obvious that software for automotive electronic systems has become a key factor not only in cost[1] and quality, but also in time-to-market of current vehicles. As pointed out by Pretschner et al. [PBKS07], the characteristics of the automotive domain lead to a wide spectrum of research challenges like inter alia:

- System models and techniques that enable semantics-preserving transformation and modification of models and code

[1] Overall costs contain not only expenditures for development and manufacturing, but also costs for maintenance, which are highly affected by the vehicles' large number and long life-cycle.

- Design methodologies at different levels of abstraction that address the heterogeneity of the systems involved as well as the compatibility problem
- Design and coding practices that lead to reusable code and thus to a reduction in complexity and costs
- Communication middleware for highly distributed heterogeneous systems

Therefore, it was self-evident that major automotive manufacturers strove for a standardization not only of software modules deployed within their vehicles, but also for an appropriate software engineering process. In 2002, automotive manufacturers, suppliers, and tool developers jointly founded the *Automotive Open System Architecture* (*AUTOSAR*) consortium [HSF+04] to specify those required standards.

The main aims of *AUTOSAR* are:

- Increase the quality of automotive software, its maintainability, and its scalability
- Optimize costs and time to market

To achieve these aims, the well established paradigm of Component Based Software Engineering (CBSE) has been adopted for the automotive domain: Applications are assembled across product lines by connecting prefabricated trusted building blocks, so called commercial-off-the-shelf components, which interact by their connected interfaces at run-time.

In fact, interaction implies inter-component communication that strongly depends on the physical system structure and the components' deployment scenario. In distributed heterogeneous systems the process of interaction can become rather complicated and difficult to handle. It is good practice to keep the complex and error-prone interaction logic separated, if possible hidden, from the application components. Therefore, component based architectures typically utilize communication middleware, to cope with the process of distributed interaction in a transparent way. Application components interact directly (at least seemingly), irrespective of their deployment scenario, by utilizing transparent communication middleware. Hence, application components do not have to provide any implementation for means of distributed interaction. This fact keeps them focused on their actual purpose, thereby

staying small in size and less error prone. However, generic middleware, like e.g. middleware for the *CORBA* Component Model [OMG06], has to provide implementations for all types of interaction that could occur within any exerting component architecture. Thus, this kind of middleware typically tends to be heavy-weight and often monolithic software.

Due to the large number of manufactured automotive electronic systems, the price per unit has to be kept as low as possible. As a consequence, automotive embedded systems are inherently resources constrained and thus do not meet the resource requirements of heavy-weight generic middleware. However, CBSE has been well accepted within the automotive domain and hence, demand for light-weight resource saving component middleware arose. By adopting CBSE for the automotive domain, the *AUTOSAR* standard extensively addresses the issue of customizable standardized component middleware.

AUTOSAR provides a precise specification for a standardized component and communication middleware. Its logic abstraction is called *Virtual Function Bus* (*VFB*) and its functionality is accessible only via the so called *Run-Time Environment* (*RTE*). All application components interact in a location transparent way only through the *RTE*. In addition, the *RTE* provides all infrastructural services implemented within the underlying *Basic Software* (*BSW*) module. The *Basic Software*—the implementation of the *AUTOSAR* middleware and the operating system—is specified as layered software that can be trimmed with respect to application requirements in order to save valuable system resources. Trimming is done by specifying various compile-time switches[2] to exclude middleware layers with unused functionality. Although this proposed approach targets at economizing system resources, it turns out to be suboptimal: Manual management of often undocumented, proprietary, and vendor specific compile-time switches is extremely error-prone. Minor changes at application level can lead to major changes in middleware functionality, requiring detailed knowledge of middleware internals from application developers and system integrators. In addition, only coarse-grained middleware customization can be achieved due to the coarse-grained modularization of the *Basic Software*.

The research presented within this thesis aims at an improvement of *AUTOSAR*'s middleware in terms of reusability, maintainability and potential for application specific optimizations with respect to system immanent

[2]These switches are, for example, preprocessor "defines" for conditional compilation in C source code.

resource constraints. Following the same advisement that has been taken into consideration by the *AUTOSAR* consortium for automotive applications, the required improvements can be achieved by applying the component paradigm to *AUTOSAR*'s middleware itself. However, this idea leads to two major research questions, covered within this thesis:

1. **Is it possible to apply the component paradigm not only at the application level but also to the *AUTOSAR* component middleware, especially to the communication subsystem, itself?** All benefits of CBSE would become available not only at application level but also within the middleware, including better quality, reusability and maintainability, and higher cost efficiency.

2. If question one can be answered with yes, the second question unfolds: **Can component based, application specific, and hence custom-tailored *AUTOSAR* middleware be automatically synthesized from application models and prefabricated middleware components?** A middleware implementation that is optimal in terms of resource usage for a specific network node provides only services required by application components deployed on this particular node. If component based middleware that is optimal in the above mentioned sense could automatically be generated, overall software quality would increase whereas costs for development and maintenance would decrease. In accordance to the component paradigm, middleware components could be provided by third parties (e.g., communication middleware components by bus system manufacturers), thus relaxing the application developers' know-how requirements.

1.2 Approach

As described above, this thesis aims at an improvement of *AUTOSAR* middleware by applying the component paradigm at the middleware itself. For that purpose, several aspects of automotive software engineering are taken into account:

1. ***AUTOSAR***: The standard's component model and the prescribed software engineering methodology are analyzed, to understand not only all provided mechanisms of component interaction and communication, but also the related engineering process.

2. **Model Driven Development**: Models provide different views of a specific application. With respect to *CBSE* and *AUTOSAR*, model artifacts that contain information regarding communication middleware functionality are identified within *AUTOSAR* compliant application models.

3. **Middleware Architecture**: To design component based communication middleware for *AUTOSAR*, it is mandatory to analyze the standardized layered *AUTOSAR Basic Software* and the *AUTOSAR Run-Time Environment*. A set of basic middleware components as well as a general component architecture representing the full *AUTOSAR* middleware functionality is identified. In addition light-weight variants of the identified component classes are specified for improved middleware optimization.

4. **Software Synthesis**: To automatically synthesize *AUTOSAR* compliant communication middleware from application models and prefabricated building blocks, a model driven process is defined. This process on the one hand extracts information from application models and system descriptions, and on the other hand automatically assembles prefabricated middleware components to form an application's custom-tailored middleware. To support model level validation of functional and non-functional requirements, as well as model checking approaches in general, the defined process conserves annotated properties and requirements over its different phases.

1.3 Contribution

This thesis contributes by introducing component based communication middleware into the *AUTOSAR* standard and by integrating the three domains of Component Based Software Engineering, Model Driven Development and *AUTOSAR* into one consistent methodology that helps to gain cost-effective, high-quality software for automotive distributed embedded systems. In detail, the contribution is twofold:

1. **The component paradigm is applied to *AUTOSAR*'s originally layered communication middleware.** Thereunto, the *AUTOSAR* communication middleware layers are horizontally sliced into blocks of related functionality. By rejoining tightly coupled blocks, irrespective of their originating layer, self-contained building blocks—the middleware components—are created. This process is on the one hand ex-

ecuted by a static analysis, the so called *Cohesion Analysis* [1][3], and on the other hand manually by domain-experts in order to get a reliable reference design for evaluation of the calculated results. The total of all component class interfaces for the identified component based middleware design resembles the full functionality and the full interface of the conventional layered middleware. However, to reduce size and resource usage of the communication middleware, the component based middleware design allows the substitution of full-fledged heavy-weight components by light-weight variants, providing only necessary, and hence reduced, functionality. In case a component's functionality is not required by the application or other middleware components, this component may even be completely omitted from the middleware binaries [2, 3]. Supplementary, a component model—the *Component Based Automotive System Software* (*COMPASS*) component model [COM07]—for the *AUTOSAR* communication middleware is specified, to prescribe how middleware components have to be used, how they may be composed, and how they interact [4]. Based on the *COMPASS* component model, reusable architectural middleware patterns are specified [5]. These patterns describe functionality of client-server and of sender-receiver communication, which are the two communication styles implemented by *AUTOSAR* within the *AUTOSAR Run-Time Environment*.

2. **A model driven component based development process for automatic middleware synthesis is specified.** This process is closely related to the Model Driven Architecture (MDA) [OMG03a], but also adheres to the *AUTOSAR* methodology [AUT08b]. The core of the specified process is a model transformation, called *Connector Transformation* [6]. It automatically transforms platform independent models (PIMs), describing the application's component architecture, into platform specific models (PSMs), containing application components and component equivalent middleware structures. The *Connector Transformation* mainly transforms PIMs into PSMs by injecting platform and communication specific middleware component architectures[4] in place of explicit connector artifacts, contained within the PIMs. The generated middleware structures depend on the set of prefabricated middle-

[3]The author's publications related to this thesis are referenced with plain numbers. For the remainder of this thesis these publications' results are used without being explicitly referenced.

[4]The term *component architecture* denotes a piece of software that is realized by a well-defined set of composed components and their connectors.

ware components, architectural middleware patterns, and the application's deployment specification [7, 8]. As a result, only utilized communication functionality of the *AUTOSAR* communication middleware is contained within the middleware binaries. The PSMs moreover reflect the system's physical structure, so one PSM is created for each system node. Hence, this synthesized communication middleware is application and node specific, and thus exhibits a smaller memory footprint than conventional *AUTOSAR* communication middleware. In addition, functional and non-functional contracts (by means of model level annotations) of middleware components are preserved during transformation, and thus are contained within the PSMs. Therefore, annotations of application components and of assembled middleware components provide a sound foundation for model checking and system validation of automotive software systems [9].

1.4 Thesis Structure

To answer the research questions issued in Section 1.2, the thesis is structured as follows:

- Chapter 2, Technological Baseline and State-of-the-Art, overviews Component Based Software Engineering (Section 2.1) and Model Driven Development (Section 2.2), and describes how these domains are related to each other. It provides basics on the key elements of CBSE and MDD, and summarizes all necessary vocabulary and notations, used within the remaining chapters.

- Chapter 3, Automotive Open System Architecture, provides an outline of the *AUTOSAR* hardware and software architecture. It is focused on the standard's sub-domains that are of specific relevance for the subject of this thesis: the component model, and all system parts related to communication middleware.

- Chapter 4, COMPASS Middleware, comprises the first major contribution of this thesis: On the one hand it defines the *COMPASS* component model that is applied to the *AUTOSAR* communication middleware, in order to solve the research challenges covered within this thesis (Section 4.2). On the other hand, the chapter identifies middleware component classes by decomposing conventional *AUTOSAR* middleware via the static *Cohesion Analysis* (Section 4.3).

- Chapter 5, Middleware Synthesis, comprises the second contribution of this thesis. It describes how to utilize augmented *AUTOSAR* application models to synthesize application specific component based communication middleware. In Section 5.2 the *COMPASS* application model, especially the model artifact of explicit connectors, is introduced, while in Section 5.3 the *Connector Transformation* is described, which performs the middleware synthesis.
- In Chapter 6, Discussion, the proposed methodology is applied to an automotive application at proof-of-concept level. Gathered results are discussed for the *Cohesion Analysis* and for the *Connector Transformation* (Section 6.1). Finally, related work that has not explicitly been mentioned within the previous chapters is discussed (Section 6.2), and a brief outlook on future work is given (Section 6.3).

Chapter 2

Technological Baseline and State-of-the-Art

This chapter provides an overview of state-of-the-art technology, representing the baseline for the contribution of this thesis. It provides fundamental concepts of component based software engineering and model driven development. As a result, a common terminology is defined for the remainder of this thesis.

2.1 Component Based Software Engineering

One substantial challenge in today's software engineering is to master the development of cost-effective and reliable software in the face of increasing size and complexity. Various approaches have been proposed over the last years, some of them made it into industry. Most of them have identified software reuse as a main candidate for "better and cheaper" software development.

Component Based Software Engineering (CBSE) is one widely accepted concept in developing cost-effective and sound software, that builds on reuse [NT95], and therefore aims at increasing software quality and productivity. In CBSE applications are built by assembling small, well-defined, and trusted building blocks, so called components. Related components are connected by their interfaces, to form a new software system with combined functionality. As components provide means of exchangeability and reusability, implicit context dependencies are strictly prohibited.

Components are subject to third party composition; therefore components of different vendors have to interact seamlessly. This can only be assured by defining a framework—a set of rules, utilities, programming abstractions, and

an infrastructure—called component model that has to be strictly obeyed and utilized by all components. A component model provides the semantic framework of what components are, and how they are constructed, composed, deployed and used [WS01, LW05]. Thus it provides the foundation for any component based application.

A component model contains at least the following specifications:

1. **Component Definition**: The component model defines, what components are. It gives a precise description of properties and constraints, which a building block has to satisfy in order to be a component.
2. **Composition Standard**: By defining how, where, and when components may be connected, the component model clearly defines rules that are of great relevance for the design process of a component based application.
3. **Interaction Standard**: At run-time, connected components interact by exchanging data or flow of control. The interaction standard defines, how data is exchanged and what happens to the flow of control within the interacting components.
4. **Infrastructure Specification**: One main issue of CBSE is to provide separation of concerns. To keep components focused on their application purpose, any implementation of infrastructural concerns, e.g. communication issues or life-cycle management, is shifted to the component middleware, which in consequence is often referred to as application server. The component middleware consequently has to provide all infrastructural services required by the application components, and thus is the vital environment for them. As a result, application components stay highly specialized to their primary purpose, small in size, and less error-prone.

Component models are usually categorized into flat or hierarchical models, or by means of component composition [LW05]. Both, the AUTOSAR component model [AUT08c, AUT08g] and the COMPASS component model (see Section 4.2), are hierarchical component models.

2.1.1 Components

In CBSE, the most common entity is called component. Unfortunately, literature shows many, very often contradictory, definitions of that term

[BDH+98, Bro98]. So the first thing to do when dealing with CBSE, is to clarify the semantic meaning of this appellation, and to provide a clear vocabulary for the domain as basis for the remainder of this thesis.

2.1.1.1 Component Definition

Within the following paragraphs, some of the most accepted component definitions are summarized to obtain a general one, which serves as foundation for all more specialized refinements, discussed and developed within this thesis.

Szyperski [Szy99] defines a component to be a composable unit that interacts by specified interfaces only:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

The definition of Heineman and Council [CH01] differs from most definitions by postulating that a component has to adhere to a component model:

> A [component is a] software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.

To obtain reliable behavior from composed software architectures, Mayer [Mey03] introduces the concept of trust in CBSE and therefore defines a component to be a trusted element of reuse:

> A component is a software element (modular unit) satisfying the following conditions:
>
> 1. It can be used by other software elements, its "clients".
> 2. It possesses an official usage description, which is sufficient for a client author to use it.
> 3. It is not tied to any fixed set of clients.
>
> [...]
> A Trusted Component is a reusable software element possessing specified and guaranteed property qualities.

For the *CORBA* Component Model [OMG06], the *Object Management Group* (*OMG*) specifies a component in a more abstract, syntactic way, related to the design phase of the development process:

> A [component is a] specific, named collection of features that can be described by an IDL [Interface Definition Language] component definition or a corresponding structure in an Interface Repository.

Finally, for their *Unified Modeling Language* (*UML*)[1], they define a component as an executable element within a system, which has an external specification in the shape of one or more provided and required interfaces, and an internal implementation, consisting of one or more classifiers that realize the component's behavior:

> A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.
>
> A component defines its behavior in terms of provided and required interfaces. As such, a component serves as a type whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics).

By taking all these considerations into account, a general component definition that is used for the remainder of this thesis can be formulated:

Components are trusted architectural elements of execution that interact only by their well-defined interfaces, according to contractual guarantees, and strictly contain no other external dependencies. Components conform to a component model, so they adhere to a composition and interaction standard, and can be independently deployed and composed without modification. Therefore, components are well suited for reuse and third-party composition.

[1] Within this thesis UML 2.1 [OMG07b] is used to describe components, component architectures, deployment specifications, and architectural templates. It provides a useful set of predefined classifiers that can easily be extended to meet domain specific requirements of architectural modeling [RMRR98].

A set of well composed components is referred to as component architecture, while the term component model denotes the framework and standards, a component has to adhere to.

Both, the *AUTOSAR* component model, described in Section 3.4, as well as the *COMPASS* component model, described in Section 4.2, adhere to this definition.

2.1.1.2 Interfaces

The behavior of a component is completely determined by its implementation and the specification of its interfaces. Following the given component definition, interfaces describe a component's points of interaction.

An interface is a set of operations—the services—and accessible memory locations, also referred to as data elements. Operations are uniquely specified by their signature, a tuple consisting of the operation's name and its ordered parameter list. Data elements are specified by a tuple containing the element's name and data type. Due to the descriptive nature of an interface, it does neither offer any implementation of its operations, nor does it provide any data element. An interface rather refers to a component implementation that provides all of that [CHJK02]. Depending on the interfaces' role in a component description, two kinds of them have to be distinguished:

1. **Provided-Interfaces** expose services implemented by and data elements contained within a component, for proper use by other components.
2. **Required-Interfaces** represent a component's need for external services or data elements, provided by other components via their provided-interfaces. Within a valid component architecture, all required-interfaces have to be connected to related provided-interfaces, while dangling provided-interfaces may occur.

Both kinds of interfaces are representable in *UML 2*, and can be expressed in various ways. Figure 2.1 depicts one sample component, named *A*, that provides services via one provided-interface, called *IAServices* and demands external services via one required-interface, called *IBServices*.

Figure 2.1 depicts component *A* and its interfaces, using the *Ball-and-Socket* notation of *UML 2*: Provided-interfaces are denoted as ball, while required-interfaces are denoted as socket. Although this notation is more intuitive

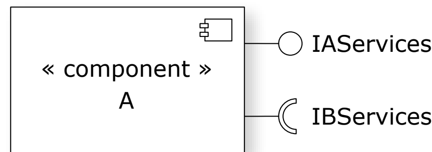


Figure 2.1: *UML 2* notation of a component

to read, less information, especially on the interfaces' properties, is visible and has to be described within extra artifacts, e.g. interface descriptions or interface contracts.

2.1.1.3 Ports

Interfaces may be exposed by a component itself, as depicted in Figure 2.1, or by a component's port, as depicted in Figure 2.2. A port is a dedicated point of interaction; it is used to group related interfaces. Ports in general are typed by all interfaces comprised within. If a port comprises more than one interface, it is referred to as a complex port. As depicted in Figure 2.2 ports in UML are denoted as rectangles, placed on the component artifact's border.

Ports are typically used for

- **Interaction Modeling**: When describing and verifying run-time behavior of composite structures, the sequence of interface invocations plays an important role. As ports are used to group interfaces, it is feasible to bind interface states [Sel99]. Interface states are the current phase of an invocation sequence of a specific port.
- **Logic Grouping of Interfaces**: Grouping semantically related interfaces increases a model's comprehensibility. All interfaces exposed by one port are of concern for the same—the port's—purpose of interaction.
- **Model Abstraction**: As ports are typed by their interfaces, they can be used as placeholder for all the interfaces they comprise. In this way, models with a higher level of abstraction like the architectural patterns defined within Section 5.3.1 can be created.

2.1.2 Connectors

Composing components connotes joining related provided- and required-interfaces. The junction between component interfaces or ports is called connector. At run-time, connected components interact through their associated interfaces via joining connectors only. While connectors in local component architectures typically represent simple issues of control- and data-flow, they become hot-spots of interaction within distributed, heterogeneous systems.

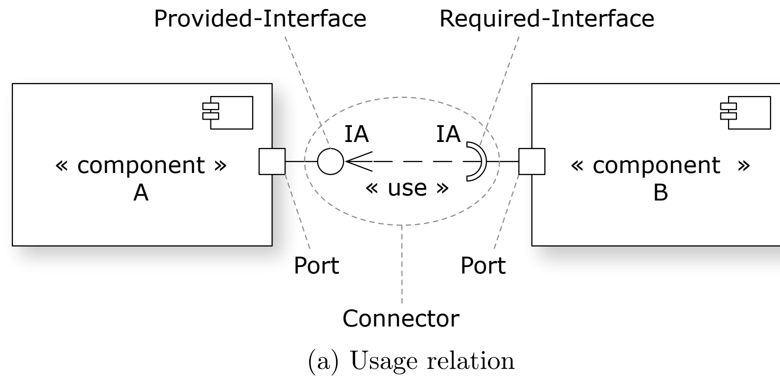
UML provides various ways to express component composition with connectors. Figure 2.2 shows the three most common *UML* notations for component connectors by a simple example: Component *A* provides services by a provided-interface of type *IA*, while component *B* requires at least one service from that interface, thus it comprises *IA* as required-interface. All three notations, depicted in Figure 2.2a, 2.2b, and 2.2c, show the involved components with their interfaces and ports, but use a different syntax[2] for the same connector. However, the port connector, depicted in 2.2c, semantically differs from the others. A port connector provides a higher-level of abstraction of a set of connected interfaces. It is typically used to simply express that two components are connected for a specific purpose, whereas detailed information on interfaces is not of interest.

1. **Usage Relation Notation**: A component that requires external services by comprising a required-interface, makes use of that services, provided by another component via a provided-interface. Figure 2.2a depicts this fact by connecting the associated required- and provided interfaces with a usage-relation, represented as dotted arrow. This notation lays stress on usage and, as e.g. in $AUTOSAR$, on the existence of a communication channel, also symbolized by the dotted line of the arrow.

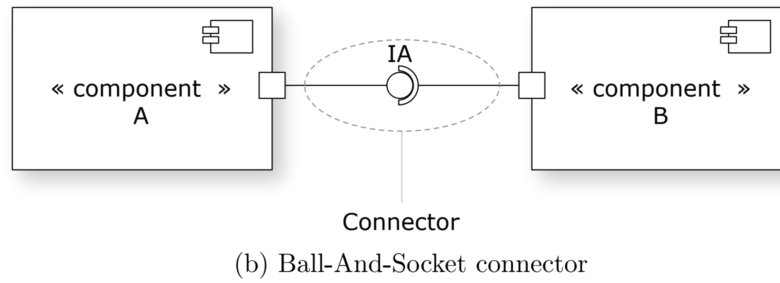
2. **Ball-And-Socket Notation**: A short-cut to the usage-relation notation is shown in Figure 2.2b. The required-interface and its associated provided-interface is directly joined into one symbol. Due to its simplicity, this notation is widely used to represent component dependencies. Within this thesis, this notation is mainly used for local connections within middleware architectures, as no "external" communication channel is used within that connections.

3. **Port Notation**: Ports, as described in Section 2.1.1.3, may represent a component's points of interaction at a higher level of abstraction. They

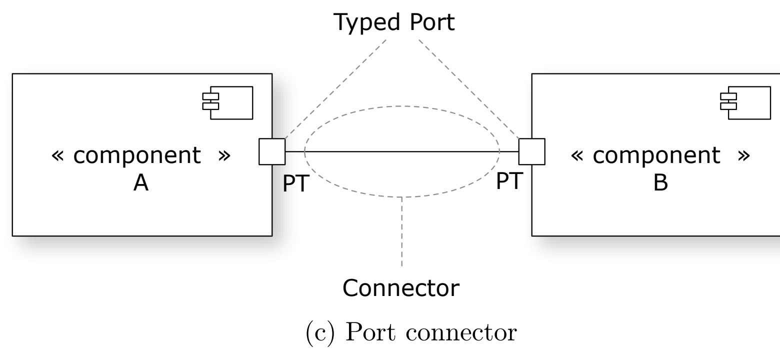
[2] *UML* provides a graphical notation for describing software models. Model elements are represented by distinct shapes—the syntax for the elements.



(a) Usage relation



(b) Ball-And-Socket connector



(c) Port connector

Figure 2.2: *UML* connector notations

comprise a well defined set of interfaces that gives them a unique type. As two components are often connected by multiple related interfaces, it is often expedient to omit the interface artifacts and simply wire the matching ports of associated components. A well known example is that of a call-back mechanism, which requires at least one provided- and one required-interface for each participant. The "wire" is symbolized by a simple line, connecting the corresponding ports, as shown in Figure 2.2c.

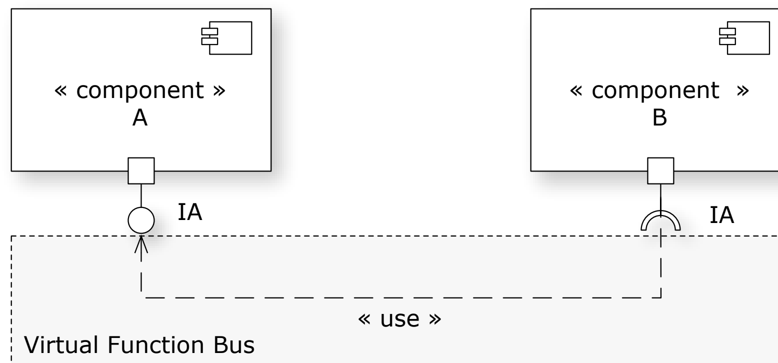
2.1.3 Component Middleware

Due to the concept of separation of concerns within component based software—components have to deal with their business purpose only—all infrastructural requirements are provided by the component model's middleware. Therefore, the middleware is the vital environment for each application component. It provides means of life-cycle management and component execution, communication and interaction services, and many other features, required by the application components to fulfill their task.

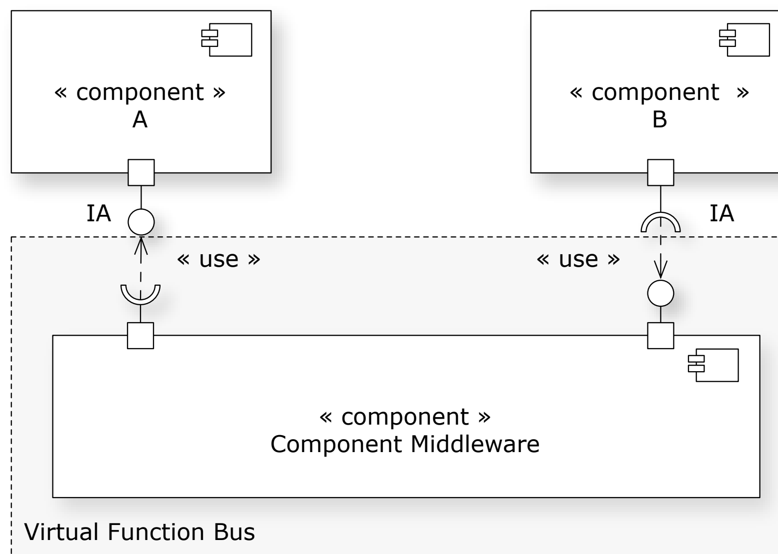
In typical component models, like the *CORBA* Component Model (CCM) and Microsoft's *Component Object Model* (COM), application components are embedded within a component container, which provides an abstraction of the component model's middleware to the application components. Although related components are seemingly connected to each other via their connectors, in reality they are connected to the middleware container that provides a virtual function bus (VFB) in a transparent way. Figure 2.3 shows two components named A and B, where B requires services from A. Sub-figure 2.3a shows the components' point of view: they seem to be directly connected over the *VFB*. Sub-figure 2.3b shows the physical view: the components are connected to the component container that manages the component interaction.

Within existing (but also proposed) component middleware implementations, a wide set of architectural paradigms is used. This fact results from the variety of middleware purposes, but also from the great amount of system properties. Many of these properties are domain- and/or application-specific, and have to be taken into consideration, when designing middleware [KG99, ICG07].

Two common architectural styles are of relevance for this thesis: the layered and the component based design.



(a) Logical view



(b) Physical view

Figure 2.3: Virtual function bus (VFB)

1. **Layered Middleware**: Layered middleware, like proposed in [AUT08a, MSM06, Sch02], is constructed as a horizontally layered software stack. Lower layers are concealed by upper ones. Therefore, each layer interacts with adjacent ones only. Application components reside on top of this software stack, and hence are connected to the stack's top-layer.

   Even though this type of architecture is modular in terms of layers, intra-layer modularity is hardly possible. The middleware itself is typically deployed as monolithic block, thus customization or middleware adaption usually requires a full rebuild. Another disadvantage of strictly layered middleware is that services from bottom layers can not be directly accessed from the application components, and thus have to be delegated through all intermediate layers. These forced service delegations cause unnecessary run-time overhead, which offers space for optimization especially in resource constrained domains like embedded systems. However, due to the well understood mechanisms in layered software, and due to easy-to-understand designs, this middleware type still plays an important role in CBSE.

   The conventional *AUTOSAR* component middleware is specified in a layered style as described in Section 3.3. Its top-most layer is called *Run-Time Environment*.

2. **Component Based Middleware**: Component based middleware, sometimes also referred to as modular middleware, like described in [PCCB00, LQS05], is designed in accordance to the component paradigm. Middleware functionality is divided into functional blocks that contain closely related, often tightly coupled functionality only. The implementation of each block's functionality represents one specific middleware component. Middleware components may seamlessly interact with each other, but also with application components. However, a container component is often used to provide functionality of interface adaption between generic middleware components and user-defined application components.

   Due to the component based design, middleware of this type can be reconfigured, reused and adapted to altering requirements without a full rebuild. Service delegations across multiple layers, which often arise in strictly layered software architectures, are omitted, as related middleware components may be arbitrarily connected without restrictions in terms of layers.

   In addition, component based middleware can be assembled automati-

cally according to application demands, and thus can provide custom-tailored, resource efficient middleware.

2.2 Model Driven Development

When building component based applications, software is constructed by assembling prefabricated components. These component architectures reflect the applications' functional decomposition, and thus provide a high level view of the developed software. Components are often also hierarchically composed architectures themselves, hiding their internals behind published interfaces. Hence, a component architecture's level of abstraction can successively be lowered by refining the view's granularity, thereby revealing internal structures of assembled building blocks. Each level of abstraction provides specific architectural details of the represented application. This fact can be capitalized on within a development process by introducing highly specialized views, corresponding to expert knowledge of the views' specific domain. In addition, standardized architectures and building blocks enable production-line like development cycles. Consequently, when developing component based software, a proper engineering methodology is mandatory to reach CBSE's goals of better and cheaper software. A well suited development process has to be formalized and wherever possible unified. It has to cope with various levels of abstraction and their correlation, and also has to provide support for software- and architectural reuse [Pas02].

One approach that can perfectly be adopted for CBSE is Model Driven Development (MDD). In MDD, system descriptions, so called models, are used to obtain a coherent total representation of the software's characteristics. Models are the focal points of a model driven development process and are subject to specification, refinement and manipulation. By applying model transformations, models can be transformed into new typically more specific ones. Transformations can also be used to extract model inherent information from source models. To subsume, a MDD process is characterized by the activity of modeling a software system and by transforming the specified models, to finally gain the required product.

A common definition of the term model is provided in [KWB03]:

> A model is a description of (part of) a system written in a well-defined language.

> A well-defined language is a language with well-defined form (syntax) and meaning (semantics), which is suitable for automated interpretation by a computer.

The first definition assesses a model to be a representation, most times an abstraction, of a system. It is typically simplified and focused on a specific subset of the system's features. The second definition more precisely states how a model has to be specified. The assumption, that a model has to be automatically interpretable by a computer, seems rather rigorous at the first look and is not demanded by most model driven approaches. Nevertheless, it is good practice, since software development, and thus MDD, is a computer aided process that relies on automation to exploit its full potential.

One of the precursors of MDD is the OMG, who defined the Model Driven Architecture (MDA) framework [OMG03a], which has become rather popular within software industry. MDA specifies concepts and languages[3], and the relation between them, but does not define a specific development process. In fact, any favored process can be adopted and adapted, if it meets the requirements of MDA: models and transformations at various levels of abstraction have to play a central role.

Within the following sections we revisit the most important aspects of MDD and MDA that are used within this thesis.

2.2.1 Viewpoints and Models

To express the level of abstraction, MDA utilizes so called views [IEE00]:

> A viewpoint model or view of a system is a representation of that system from the perspective of a chosen viewpoint.

To render this definition more precisely, the term viewpoint is also clarified [OMG03a]:

> A viewpoint on a system is a technique for abstraction using a selected set of architectural concepts and structuring rules, in order to focus on particular concerns within that system. Here 'abstraction' is used to mean the process of suppressing selected detail to establish a simplified model.

[3] MDA requires models to be expressed in a *Meta Object Facility* (*MOF*) based [OMG03b] language.

Following the given definitions, models are descriptions—also referred to as views—of a system that are typically focused on specific system aspects. The specific aspect covered within a model, strictly depends on the model's viewpoint.

OMG's MDA proposes three distinct viewpoints on a system, which are of great importance in model driven development:

1. **Computation Independent Viewpoint**: By focusing on the system's environment and its requirements, the computation independent viewpoint hides details of the system's structure and processing. This viewpoint's main purpose is to separate the fundamental logic of a system, e.g. process- or business-models, from its technical specification.
2. **Platform Independent Viewpoint**: The platform independent viewpoint aims at the operation of a system while disregarding platform specific details. A platform is considered to be a set of subsystems and technologies, which provide a coherent set of functionality through interfaces and specific usage patterns. Platform independent views expose aspects of a system that are shared between distinct target platforms, and are typically specified in general-purpose modeling languages such as UML.
3. **Platform Specific Viewpoint**: The platform specific viewpoint is focused on the use of a specific platform. It exposes aspects of a system that can not be shared between different target platforms.

These distinct viewpoints help to gain a clear separation of concerns within a model driven development process, and also lead to a better comprehensibility of the overall system.

In accordance to the viewpoints' aspects, MDA proposes four types of models that are all used within the following chapters. Thus, their purpose is summarized here:

1. **Computation Independent Model (CIM)**: A description of a system from the computation independent viewpoint is called computation independent model. A CIM describes the situation and the context in which the system will be used. Therefore, it includes no details of a system's structure. However, a CIM typically not only serves as aid to understand a problem, but also as source of system specific vocabulary for use in other models. CIMs are often referred to as domain models, and play an important role in bridging the gap between domain

experts on the one hand, and experts in design and construction of system artifacts, on the other hand.

2. **Platform Independent Model (PIM)**: A view of a system from the platform independent viewpoint is called platform independent model. A PIM exposes aspects of a system that primarily deal with logical and structural concerns, independent of any implementation technology. Within this thesis PIMs are used to specify component architectures at application level. Matters of distribution or heterogeneity are not of concern within this models.

3. **Platform Specific Model (PSM)**: A platform specific model is a view of a system from the platform specific viewpoint. A PSM contains aspects of a system that are directly related to a specific platform, the associated implementation technology. It augments relevant aspects of a PIM with details on how the system uses a particular type of platform.

4. **Platform Model**: A platform model specifies technical concepts, system parts, and provided as much as required services of a specific platform. It also provides concepts on how the platform has to be used by an application, and how this usage has to be described within a PSM. Platform models are a vital source of information for the model transformation, specified within this thesis.

By utilizing distinct viewpoints, a system is described by multiple models of various types. As these models represent different abstractions of one system, they are related to each other. Well-defined relations between models of the same system may be used to (sometimes even automatically) translate models of one type into another. To aid this process of translation, models may be annotated with marks. Within this thesis, marks are attached to elements of PIMs (connectors in component diagrams), to guide proper translation of PIMs to PSMs. In MDD, the process of translation is referred to as transformation.

2.2.2 Transformations

The real value of MDD, especially of MDA, results from its ability to transform models of distinct viewpoints into each other. Platform independent models can be transformed to platform specific models. Platform specific models can be translated to code. Traditionally, most of this work has been done by hand. To automate this process, model transformations have been

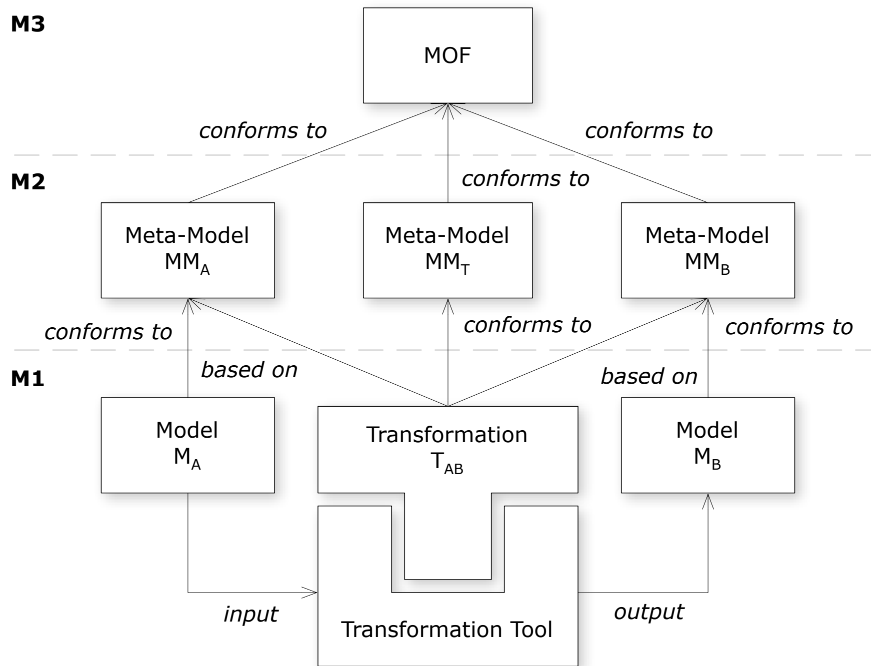


Figure 2.4: Model transformation

subject to research and development over the last years. Automated transformations impose big potential in reducing costs, and in increasing software quality in MDD, as they reduce the amount of error prone human intervention.

Transformations are typically specified as set of rules, denoted in a transformation language like OMG's *Query/View/Transformation* (*QVT*) [OMG07a] or the *Atlas Transformation Language* [ATL06, JK06]. These rules are applied to a source model by a transformation tool to generate a target model.

Figure 2.4 depicts an MDA compliant model hierarchy of the transformation step: A transformation T_{AB}, executed by a transformation tool, translates the source model M_A into a target model M_B. The transformation itself is a model[4], therefore M_A, M_B, and T_{AB} are elements of model level $M1$. While M_A and M_B conform to their meta-models MM_A and MM_B respectively, T_{AB} not only conforms to its meta-model MM_T, but is also based on the

[4]Source code is also considered to be a model of an executable program.

meta-models of the source and the target model, as it has to cope with both of them. All three meta-models are elements of model level $M2$ and finally conform to the Meta-Object Facility at model level $M3$.

Depending on the meta-model, to which the source and target models comply to, two types of transformation can be distinguished:

1. **Endogenous Transformation**: Any transformation, translating a source model into a target model that complies to the source model's meta-model, is called endogenous transformation. For an endogenous transformation, M_A and M_B within Figure 2.4 both have to comply to the same meta-model ($MM_A = MM_B$). Within this thesis, endogenous transformations are used to translate UML models (PIMs) into other UML models (PSMs).
2. **Exogenous Transformation**: If a transformation translates a source model into a target model that complies to a different meta-model, the transformation is called exogenous transformation. If M_A and M_B in Figure 2.4 comply to distinct meta-models ($MM_A \neq MM_B$), the transformation is an exogenous one. This type of transformation is used within this thesis primarily when generating glue code, configuration files, and build control files.

2.2.3 MDA Development Phases

With respect to the previously defined views, a typical model driven development cycle for a software application as outlined by MDA contains following phases:

1. **Specification of CIMs**: System requirements and domain specific enabling technologies are described within CIMs that primarily describe the application's context, and define a common vocabulary for all forthcoming models used within the application's development cycles. Requirements of the computation independent models later on may be mapped to platform independent but also to platform specific models and vice versa.
2. **Specification of PIMs**: Platform independent models are developed in accordance to the application's domain models (CIMs). They mainly provide information on the application's logical structure, but also an architectural overview of the software system under development.

3. **Platform Specification**: The underlying system's technical concepts, like available interfaces, usage patterns, architectural properties, but also physical parts, and available resources are specified within platform specifications.

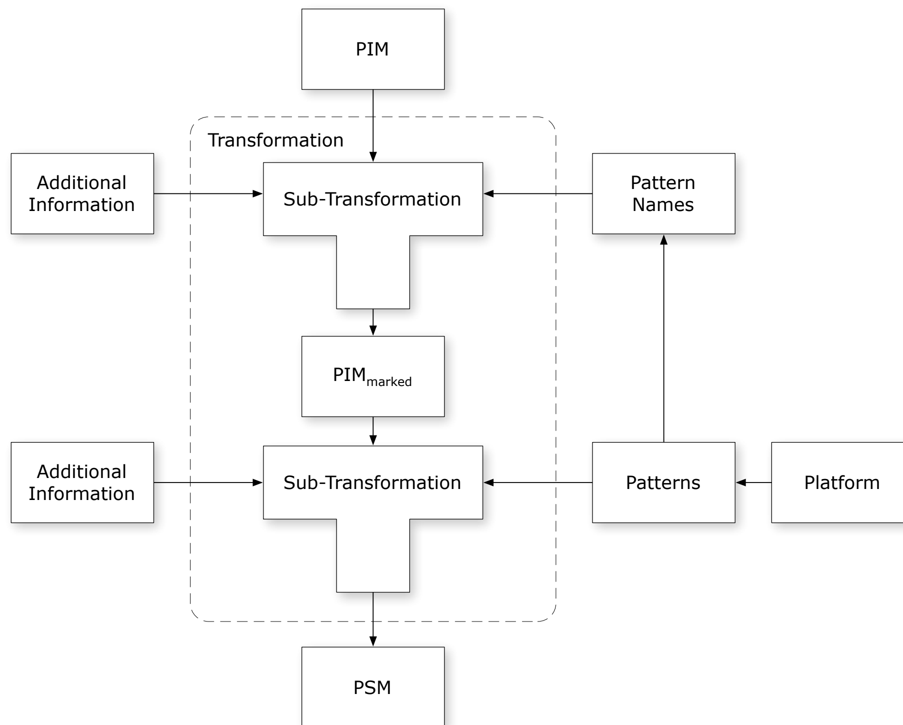


Figure 2.5: Guided transformation

4. **Transformation of PIMs to PSMs**: The heart of an MDA compliant MDD process is the transformation of PIMs to PSMs, or in case of model-to-code transformations the transformation of PIMs or PSMs into source code. Figure 2.5 depicts a guided transformation as proposed in [OMG03a]: The main transformation is decomposed into two sub-transformations.

   (a) The first sub-transformation (the upper one in Figure 2.5) translates a PIM into another PIM, by adding platform specific marks—the platform related pattern names—to the PIM. The so gained marked PIM is still platform independent in sense of its original elements and design, but it contains clues—the marks—that help

to associate the elements of the PIM with platform specific counterparts or patterns. Both sub-transformations in addition allow the specification of information like Quality-of-Service (QoS) attributes or architectural styles.

(b) The second sub-transformation (the lower one in Figure 2.5) finally translates the marked PIM into a PSM, utilizing platform specific patterns and usage descriptions from the platform model.

5. **Transformation of PSMs to Code**: A final step within a model driven development process is that of code generation. Typically, PSMs are translated into deployable executable code. This transformation is most times carried out by translating PSMs to source code that in succession is compiled and linked into an executable binary. However, code can also be generated directly from PIMs by model-to-code transformations, or via model-to-source transformations including the additional compiler- and linker-runs.

To improve a development process following this work flow, the process has to be cyclic for iterative refinement. This so called round trip engineering requires transformations to be bijective. Changes within a target model have to be transformed back to the source model to guarantee model consistency. Otherwise those changes would get lost, when transforming the source model during the following iteration of the development cycle.

2.3 Summary

Outlining the state-of-the-art of component based software engineering and model driven development, this chapter provided the basic concepts and vocabulary that come to use within this thesis. The term component, as much as terms related to components, like connector and middleware, were discussed; based on these given definitions a component model will be introduced in Chapter 4. Additionally, concepts of model driven development were summarized; the model driven middleware synthesis that is described in Chapter 5 was developed in line with the concepts of OMG's MDA.

The next chapter provides a short introduction to the *Automotive Open System Architecture* (AUTOSAR), which is the target domain of this thesis. The chapter's main focus is set to component based software engineering and model driven development as specified by *AUTOSAR*, to provide the background for the contribution of this thesis.

Chapter 3

Automotive Open System Architecture

Based on the technological baseline, provided in Chapter 2, this chapter gives an introduction to the *Automotive Open System Architecture*. Besides a general overview, the main focus is set on aspects of *AUTOSAR* that are of interest for the contributions of this thesis: the software architecture of *AUTOSAR* applications, and the architecture of *AUTOSAR* system software, especially of its communication services and middleware.

3.1 History of AUTOSAR

Driven by an increasing number of requirements of innovative applications, automotive electronic systems have reached a level of complexity that requires a technological breakthrough in order to manage them cost-effectively and at high quality. This breakthrough can be achieved with a standardized engineering process similar to that of production lines that besides are very familiar to the automotive industry since Henry Ford. By building complex software from standardized components, and by using standardized tools within a standardized process, better and cheaper automotive software can be realized.

To develop an open and standardized automotive software architecture and adequate development paradigms for automotive electronic systems, the *AUTOSAR* consortium was jointly founded by automobile manufacturers, suppliers, and tool developers in 2002 [HSF+04]. Preparatory discussions on the common challenges and objectives were held by *BMW*, *Bosch*, *Continental*, *Daimler Chrysler*, *Volkswagen*, and *Siemens VDO* in August 2002. A

joint technical team was set up in November of that year, to establish a technical implementation strategy. The partnership between the *AUTOSAR* core members was formally signed off in July 2003. *Ford Motor Company*, *Peugeot Citroën Automobiles S.A.*, *Toyota Motor Corporation* and *General Motors* joined the consortium as core partners later on. Today the *AUTOSAR* consortium consists of 9 core partners, more than 50 premium members, more than 80 associate members, and 7 development members[1]. The main objectives of *AUTOSAR* are to increase the quality of automotive software, its maintainability, and its scalability, to support usage of Commercial-Off-The-Shelf (COTS) components across product lines, and finally to optimize costs and time to market.

The consortium so far created detailed specifications in several releases: In December 2006, *AUTOSAR* phase I was finalized with the release of version 2.1 of the *AUTOSAR* standard. At the moment, phase II is in progress, and is scheduled to last until end of 2009, with the finalization of the *AUTOSAR* standard, version 4.0. The actual standard under release is that of version 3.1, finalized in June 2008. Most of the *AUTOSAR* specific assumptions within this thesis are based on version 2.1 but also comply to version 3, the most actual version while writing.

3.2 Hardware Architecture

The hardware architecture of automotive systems can be viewed at different levels of abstraction.

On the highest level of abstraction, the *system level*, an automotive system consists of a number of networks interconnected via gateways. In general these networks correspond to the different functional domains that can be found in today's cars (i.e., chassis domain, power train domain, body domain).

The networks themselves comprise a number of electronic control units (ECUs), which are interconnected via a communication media. The physical topology used for the interconnection is basically arbitrary. However, bus, star, and ring topologies are the most common topologies in today's cars. The *network level* represents the medium level of abstraction, used for the specifications of automotive electronic systems.

[1] Numbers were extracted from the *AUTOSAR* web site at `http://www.autosar.org/` last visited on 11.01.2009

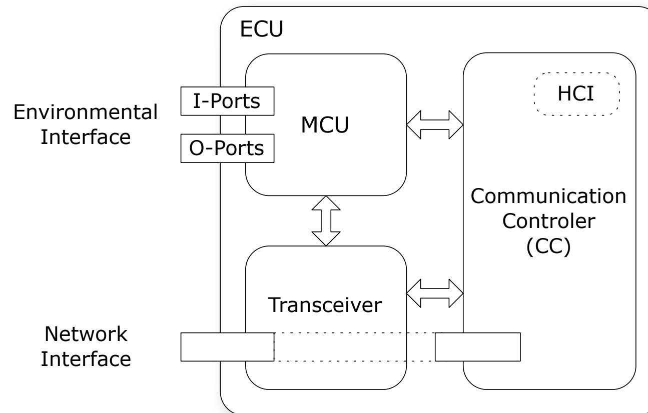


Figure 3.1: ECU design

On the lowest level of abstraction, the so called *ECU level*, the major parts of an ECU are of interest, like exemplarily depicted in Figure 3.1. An ECU comprises one or more micro controller units (MCUs) as well as one or more communication controllers (CCs). In most cases, exactly one MCU and one CC are used to build up an ECU. In order to be able to control physical processes within the car (e.g., control of the engine's injection pump) the ECU's MCU is connected to actuators via analogue or digital output ports. To obtain environmental information, sensors are connected to the MCU's analogue or digital input ports. This interface is referred to as the ECU's environmental interface. The CC facilitates the physical connectivity of the ECU to the respective network. This interface of an ECU is called network interface. The CC, as depicted in Figure 3.1, may contain a host controller interface (HCI) that provides dual ported memory. This memory is used to buffer data, which is received by the CC from the network interface for the MCU, or sent to the network by the MCU via the CC. In addition, an ECU may contain a so called transceiver. This building block is used to encode and decode logical bits into their physical representation within network communication, and thus is connected to the MCU as much as to the CC.

3.3 Software Architecture

The *AUTOSAR* software architecture, as depicted in Figure 3.2, specifies a rather strict distinction between application software and system soft-

ware, also referred to as *Basic Software* [AUT08a]. The *Basic Software* is organized as layered software architecture, which provides functionality like communication protocol stacks for automotive communication protocols (e.g., FlexRay [MHB+01, FHHW03]), the operating system, and diagnostic modules. The application software is component based, and comprises all application specific software items like control loops and sensor/actuator interaction. Consequently, the basic/system software provides the basis, on which the application software is built upon.

The so-called *Run-Time Environment* (RTE) [AUT08g] provides the interface between application software components and the basic software. It implements means of execution, coordination, and interaction patterns, but also provides interface adaptation for application components and the *AUTOSAR Basic Software*. The *Run-Time Environment* maps application specific communication requirements to fundamental communication facilities within the *Basic Software*. It is generated at compile-time to omit unused functionality within the system's executables, hence providing means of optimization for the *AUTOSAR* middleware. However, this type of resource-aware optimization is performed at a rather high level of the system software. Thus, it offers coarse-grained optimization facilities only, compared to the approach discussed within this thesis.

The term *communication middleware* as used within this thesis maps to those parts of the *Run-Time Environment* and the *Basic Software* that are related to interaction and communication.

3.3.1 Application Software Architecture

AUTOSAR compliant application software consists of application software components that are ECU and location independent, and sensor-actuator components that depend on specific ECU hardware, and therefore are location dependent. Whereas instances of application software components can easily be deployed to arbitrary ECUs, instances of sensor-actuator components have to be deployed to a specific ECU for reasons of hardware dependency. Deploying multiple instances of the same component on one single ECU is supported by the *AUTOSAR* component model. However, race conditions and data conflicts that might occur due to multiple instantiation or reentrancy, have to be solved by the component developer and are not within the scope of *AUTOSAR* middleware.

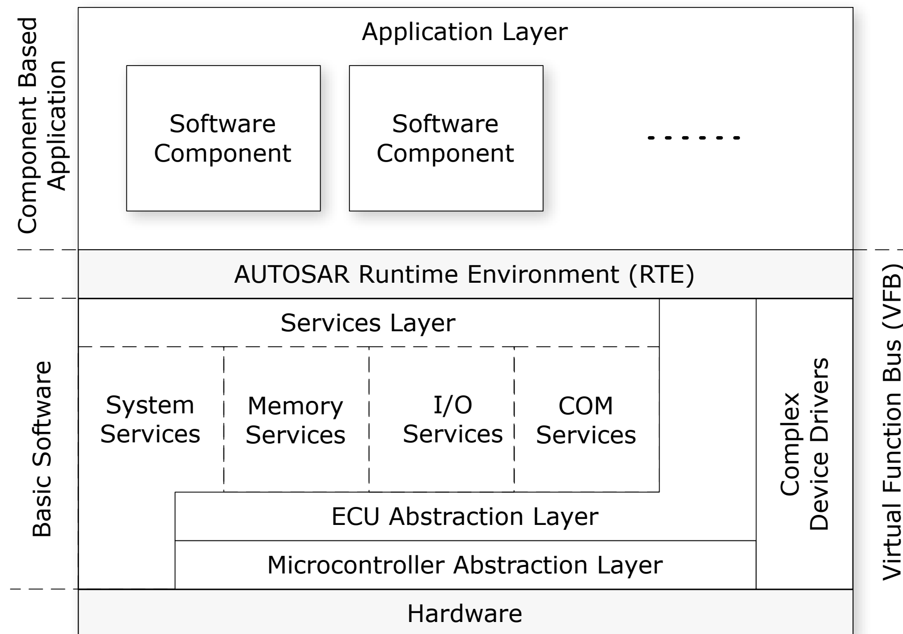


Figure 3.2: AUTOSAR software architecture [AUT08a]

Application software components as well as sensor-actuator components are interconnected via so-called connectors. These connectors represent the components' run-time interaction, like the exchange of signals[2] or remote method invocations among the connected components. A more detailed description of the *AUTOSAR* component model is provided in Section 3.4

3.3.2 System Software Architecture

Due to the designated separation of concerns *AUTOSAR* software components rely on component middleware as described in Section 2.1.3. However, *AUTOSAR* middleware does not support dynamic operations like service discovery or late binding, as automotive applications are statically typed and bound for safety reasons. Nevertheless, the middleware has to provide a basic platform for the execution of application components. In *AUTOSAR*, this functionality is provided by a layered architecture of system software modules. Figure 3.2 gives a coarse overview of the major categories of these modules.

[2]In *AUTOSAR* the term *signal* denotes any type of variable.

1. **System Services**: The *System Services* module encompasses all functionality that provides standardized (e.g., operating system, timer support, error loggers) and ECU specific (e.g., ECU state management, watchdog management) system services and library functions.

2. **Memory Services**: The *Memory Services* module comprises functionality that facilitates the standardized access to internal and external non-volatile memory for means of persistent data storage.

3. **I/O Services**: The *Input/Output Services* module contains functionality that provides standardized access to sensors, actuators and ECU on-board peripherals (e.g., D/A or A/D converters).

4. **Communication Services**: Last but not least, the *Communication Services* module contains functionality that provides standardized access to vehicle networks (i.e., the Local Interconnect Network (LIN) [LIN06], the Controller Area Network (CAN) [ISO03a, ISO03b], and FlexRay [MHB+01, FHHW03].

It is important to mention that the *AUTOSAR* nomenclature clearly distinguishes between the *Communication Services* module, included within Figure 3.2, and the *Communication Stack*, which is depicted in Figure 3.3 for a *FlexRay* network. The *Communication Stack* provides a logic view for communication functionality provided by the *Communication Services* module, as much as by lower layers like the *ECU Abstraction* layer or the *Microcontroller Abstraction* layer.

Research for this thesis was conducted within the context of automotive systems utilizing the FlexRay time-driven bus system. Therefore, the FlexRay communication stack has been analyzed, decomposed, and re-engineered in a component based way, and hence will be described in more detail within the following paragraphs. However, the proposed approach can be applied to other communication subsystems like the CAN stack in the same way.

The internal structure of the *Communication Services* module for the FlexRay communication system is depicted in Figure 3.3. It contains following (sub)modules:

1. **Com**: The *Com* module provides signal based communication to higher layers, especially to the *RTE*. This signal based communication service is used for intra-ECU communication, as well as for inter-ECU communication. In case of intra-ECU communication *Com* mainly uses shared

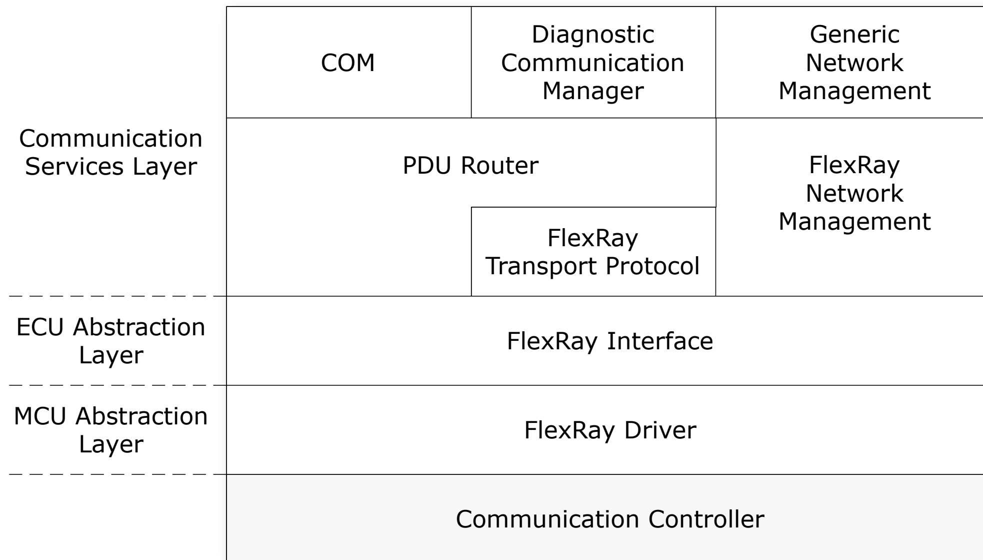


Figure 3.3: Simplified FlexRay communication stack

memory, whereas in case of distributed (inter-ECU) communication a more complex mechanism is provided: At the sender side, *Com* packs multiple signals into one Protocol Data Unit (PDU). Thereafter, *Com* passes this PDU down to a *PDU router* in order to issue the PDU's transmission via the respective network interface. At the receiver side, *Com* obtains a PDU from the *PDU router*, extracts the signals contained within, and then forwards the extracted signals to the higher software layers.

2. **Diagnostic Communication Manager**: The *Diagnostic Communication Manager* (*Dcm*) module provides services for ECU diagnosis via the communication network. The *Dcm* supports the Keyword Protocol 2000 (KWP2000) standardized in ISO/DIS 14230-3 [ISO99] and the Unified Diagnostic Services (UDS) protocol standardized in ISO/DIS 14229-1 [ISO06].

3. **Network Management**: The *Network Management* modules provide means of coordinated transition of the system's ECUs into, and out of a low-power (or even power down) sleep mode. *AUTOSAR* network management is divided into two modules: a bus system independent module named *Generic Network Management* (Nm) and a bus system dependent module named *FlexRay Network Management* (FrNm) in

case of a FlexRay bus.

4. **PDU Router**: The *PDU Router* module (PduR) provides two major services: On the one hand, it dispatches PDUs, received from the underlying interfaces (e.g., *FlexRay Interface*) to the higher layers (e.g., *COM*, *Diagnostic Communication Manager*). On the other hand, it performs gateway functionality between multiple communication networks, connected to the same ECU. The *PDU Router* forwards PDUs from one network interface to another, which may be either of same (e.g., FlexRay to FlexRay), or of different type (e.g., CAN to FlexRay).

5. **FlexRay Transport Protocol**: The *FlexRay Transport Protocol* module (FrTp) is used to perform segmentation and reassembly of large PDUs—also referred to as messages—transmitted and received by upper layers like the *Diagnostic Communication Manager*. This protocol is rather similar to the ISO TP for CAN specified in ISO/DIS 15765-2.2 [ISO04].

6. **FlexRay Interface**: Relying on frame-based services provided by the *FlexRay Driver* (see below) the *FlexRay Interface* module (FrIf) facilitates transmission and reception of PDUs. Multiple PDUs may be packed into one single network frame at the sending ECU, and in return have to be extracted at the receiving ECU. The temporal execution schedule of so called communication jobs governs when packing and unpacking of frames takes place, and when frames are handed over to or from the underlying *FlexRay Driver*. Each of these communication jobs consists of communication operations, where each is able to handle exactly one network frame including all PDUs contained within.

7. **FlexRay Driver**: The *FlexRay Driver* module provides the basis for the *FlexRay Interface module*, by facilitating frame based transmission and reception of data via a *FlexRay* communication controller.

3.4 AUTOSAR Software Components

As outlined in Section 3.3, component based software engineering is *AUTOSAR*'s development paradigm at application level. It provides clear separation of infrastructural and application concerns. Any application specific functionality is assigned to *AUTOSAR* software components in the application layer. Functionality related to the system's infrastructure has to be provided by layered software modules beneath (and including) the *AUTOSAR Run-Time Environment*.

The following sections provide a description of the *AUTOSAR* component model. Due to its design it is well suited at application level, but is not applicable at *Basic Software* level. However, as the *AUTOSAR* component model specifies the requirements for its component middleware, it plays an important role for work described within this thesis.

3.4.1 Software Component Definition

An *AUTOSAR* software component [AUT08c] is a unit of execution that implements a part of the application's functionality. It may be of atomic or composed nature. However, it has to be atomic in terms of deployment. Therefore, it cannot be distributed over several *AUTOSAR* ECUs. This restriction results from the fact, that *AUTOSAR* software components must not implement communication and interaction specific logic. Application components are not allowed to directly interact with the operating system or the communication hardware, but have to interface to the *Virtual Function Bus*. In consequence, software components are transferable, and hence may be assigned to their final deployment location very late within the development process.

A software component is described by its ports. Ports are well-defined points of interaction that are characterized by interfaces. An interface may be provided or required, thus in *AUTOSAR* a port may be of class required (R-port) or of class provided (P-port). One port comprises exactly one interface. Hence, this interface is called port interface. An interface can either be a client-server interface—defining a set of operations that can be invoked—or a sender-receiver Interface—allowing the usage of data-oriented communication mechanisms over the VFB (see Section 3.4.2).

The *AUTOSAR* component definition presented so far, maps to the component definition provided in Section 2.1.1. The restrictions described above form a specialization of the common definition provided there. The following characteristics of *AUTOSAR* components describe, how the components are embedded within the standard's environment, and how they are executed. These properties are not explicitly mentioned within the component definition in Section 2.1.1. However, they are addressed on a general level by the term "elements of execution".

Atomic *AUTOSAR* software components generally consist of a set of runnable entities. These entities correspond to implementation descriptions and behavior specifications, and are basic blocks of execution and scheduling

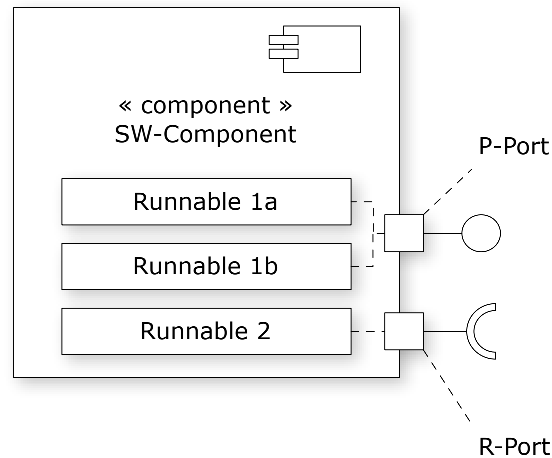


Figure 3.4: Software component and runnable entities

for the *RTE*. Figure 3.4 depicts an *AUTOSAR* software component and its internal runnable entities, whereas the entities *Runnable 1a* and *Runnable 1b* are associated with the component's P-port, and the entity *Runnable 2* is associated with the component's R-port.

AUTOSAR software components have to cope with various system states that are controlled by a *Mode Manager*, which is part of the *BSW's Services* layer. As the behavior of a software component is mainly determined by its runnable entities, mode awareness, as much as mode-change awareness, is typically implemented as configuration of runnable entities. In Figure 3.4, the P-port is associated with two entities, each for a specific set of modes. To adjust its behavior to a specific mode, a component may use one of two standardized mechanisms:

1. **ModeSwitchEvent**: By defining a specific *ModeSwitch* event, a component registers one of its runnable entities to be executed by the *Run-Time Environment* on mode change (entry and/or exit).
2. **ModeDisablingDependency**: A component's configuration specifies if a runnable entity is executed on arrival of its associated execution event (*RTEEvent*) in the system's actual mode. This mechanism is rather handy if, e.g., the system is in low-power mode and certain entities simply should not be executed to save power.

To specify a component within *AUTOSAR*, a *Software Component Descrip-*

tion has to be provided. This description is formalized within predefined Extensible Markup Language (XML) documents and contains the following information:

1. **Interface Definitions**: By specifying the component's ports and interfaces, all provided and required operations, as much as data elements are defined. *AUTOSAR* interfaces may contain operation definitions for method invocations, and data elements that are typically used in sender-receiver interaction.

2. **Infrastructural Requirements**: The infrastructural requirements specify system level services that have to be provided by *AUTOSAR* system software.

3. **Resources Needs**: The component's requirements on system resources are typically related to memory consumption and CPU usage. Resources are classified into static and dynamic ones.

   - **Static Resources**: Static resources can only be allocated by one entity. If the required amount of resources is greater than the available one, the process including the resource allocation fails. A typical example for static resources is Read-Only Memory (ROM).
   - **Dynamic Resources**: Dynamic resources can be allocated to different threads of control over time. A typical example for a dynamic resource is a CPU with respect to processing-time; different runnable entities obtain the same CPU for a specific time.

4. **Implementation specific information**: The component's internals (internal structure, runnable entities, its behavior, etc.) are denoted as implementation specific information within the component specification.

3.4.2 Composition and Interaction

AUTOSAR application components may be connected, if their interfaces are of compatible type and communication paradigm, and one is a provided interface while the other is a required interface.

A composition, specified within a component architecture at application level, is of abstract nature, and does not physically exist. Instead, components are directly connected to the *VFB*, like depicted in Figure 2.3b on page 18. The *AUTOSAR Virtual Function Bus* is a logical abstraction of all *Basic*

Software layers, involved in component interaction. Therefore, it represents *AUTOSAR*'s application specific component middleware.

At run-time, composed components interact via a concrete implementation of the *Virtual Function Bus*. The *VFB's* interfaces are provided by the *Run-Time Environment* that is generated during the development process. *RTE* generation considers configuration specific properties and the application's communication requirements [AUT08g, AUT08b].

AUTOSAR supports two communication paradigms for component interaction:

1. **Client-Server Communication**: In client-server communication, a server provides a service—an operational functionality—that is required and used by the client. The client initiates communication by requesting a service from the server, transferring a parameter set if necessary. The server receives the request, performs the operation, and dispatches a response to the client. This direction of initiation is used to categorize whether an *AUTOSAR* software component is a client or a server. A component may occupy both roles, client and server, depending on its design and realization. Client requests can be blocking—the client's thread of execution is stalled until the response from the server is received—or non-blocking, often referred to as asynchronous requests. A non-blocking client will issue a request to the server, but will not stall its thread of execution to wait for a response. Therefore, any non-blocking client that expects results from a server, has to provide a call-back interface for response delivery.

   Due to the nature of client-server interaction, its interfaces are typically procedural ones. Services are specified in terms of operations, which are identified by a unique function signature. This signature contains the operation's name, as much as number, sequence and type of its parameters.

2. **Sender-Receiver Communication**: In sender-receiver communication, a sender component emits messages (data) according to its interface specification in a non-blocking way. This information is received by one or more receiver components. In *AUTOSAR*, the sender just provides the information, while the receivers decides autonomously when and how to use this information. It is the responsibility of the communication infrastructure to distribute the information. Therefore, sender-receiver communication provides means of non-blocking, loosely cou-

pled, one-to-one and one-to-many data distribution. Sender-receiver interaction is typically one-way communication. However, in dependable automotive systems it is often feasible to implement sender-receiver communication with bidirectional data flow[3], to provide information on the communication status (e.g., receipts and acknowledgments).

Information, transmitted for interaction purpose, can be processed in two different ways within a receiver. Therefore, two types of sender-receiver interaction have to be distinguished in *AUTOSAR*:

- **State Transmission**: The main purpose of this interaction type is to distribute plain data values. Thus, this type typically implements a "last-is-best" semantic—only the last received data value is valid and accessible. Previously stored values are overwritten by the actual one. In *AUTOSAR* the term *Data Distribution* is used for this interaction type.
- **Event Transmission**: In event transmission, distributed data represents messages that denote occurred events. These messages must not overwrite previously received ones at the receiver. Therefore, receivers for this interaction type typically implement message queues.

3.4.3 Component Middleware

Automotive applications have to be statically bound, due to safety reasons. Thus, automotive component middleware does neither have to provide means of dynamic binding or run-time deployment, nor does it have to deal with service discovery, or directory look-ups. Instead, it has to be resource aware and custom tailored.

As described in [AUT08a] (also see Figure 3.2) on page 32, *AUTOSAR* does not provide an explicit component container to load components into. Instead, *AUTOSAR* software components are bound to the *Run-Time Environment* as implementation of the *Virtual Function Bus*. The *RTE* not only provides infrastructural services to the software components, but also exposes means of interaction logic, life-cycle management, and component execution, all based on functionality of the *Basic Software* layers. Therefore, when talking of *AUTOSAR* component middleware, various parts of the *Run-Time Environment* and the the *Basic Software* are addressed.

[3]The bidirectional data flow of sender-receiver communication is usually implemented at middleware level and exposed to the application level via assertions and design-time contracts.

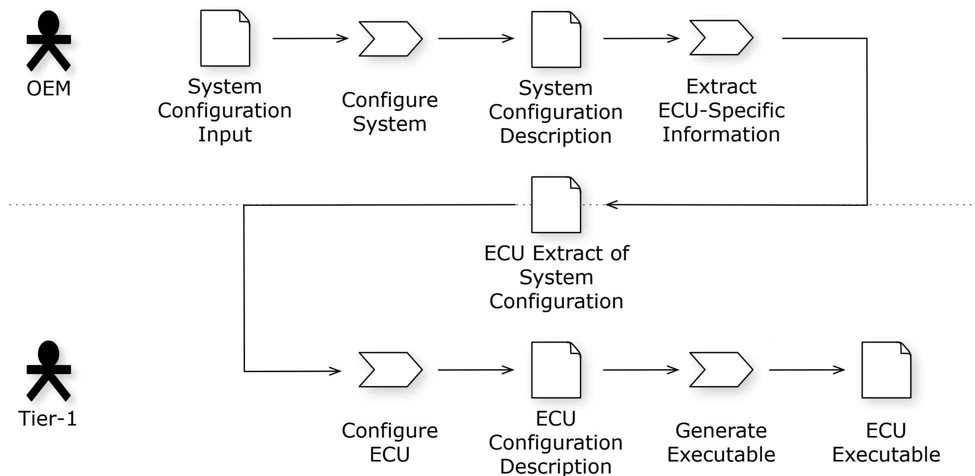


Figure 3.5: Overview of the AUTOSAR methodology [AUT08b]

3.5 Methodology

The *AUTOSAR* methodology [AUT08b] serves as guide that provides a work product flow without restrictive timing or sequence constraints of its activities. Thus, it provides information on work product dependencies and prerequisites, as much as on results of activities. The methodology is specified using OMG's *Software Process Engineering Metamodel* [OMG05] (see Appendix A).

Figure 3.5 provides a coarse outline of the application development process as specified by the *AUTOSAR* methodology. The depicted process does not include activities of application component development. These activities are within the responsibility of the Tier-1 supplier, who provides component specifications within the *Configure ECU* activity, and component binaries within the *Generate Executable* activity.

- The starting point for the prescribed process is the definition of the *System Configuration Input* work product. This architectural task identifies the system's hardware, its topology, the software components, and overall system constraints.

- The first activity, *Configure System*, mainly maps software components to ECUs with respect to resource usage and timing constraints. As a result, the work product *System Configuration Description* contains all

system information like bus-mappings, the system topology, and the component deployment specification.

- The next activity, *Extract ECU-Specific Information*, extracts specific information for each ECU within the system, and stores it in the work product *ECU Extract of System Configuration.*

Typically, all activities so far are within the responsibility of the Original Equipment Manufacturer (OEM), whereas all following activities are within the responsibility of the tier-1 suppliers[4].

- The activity *Configure ECU* adds implementation relevant information, like assembled application components, scheduling tables, assignment of runnable entities to tasks, requirements on BSW modules etc., to the system description, and in sequence creates the work product *ECU Configuration Description.*
- Finally, the activity *Generate Executable* creates an executable for each ECU, based on the *ECU Configuration Description.* This step typically involves code generation for the RTE, compilation, and linking of binaries.

3.6 Summary

AUTOSAR provides a well-defined software architecture for the automotive domain. To foster the cost-effective development of high-quality software, the *AUTOSAR* standard relies on the component paradigm for application software, and on a modular layered design for its system software. As this thesis aims at component based basic software for the *AUTOSAR* communication subsystem, this chapter provided a survey of the standard's concept of software components, as much as its layered architecture for communication middleware.

Chapter 4 in succession will define a component model for the *AUTOSAR Basic Software*. This component model finally will be used to automatically synthesize *AUTOSAR* compliant communication middleware as described in Chapter 5.

[4]The term *tier-1* denotes well known, "preferred" suppliers. *Tier-1* suppliers typically provide products and services to automotive manufacturers in very large numbers, at appealing prices or other favorable terms.

Chapter 4

COMPASS Middleware

This chapter comprises the first major contribution of this thesis. It specifies a component model for *AUTOSAR* compliant communication middleware, and defines middleware component classes, whose implementations—if properly composed—constitute an application's custom-tailored communication subsystem. Classes are identified by decomposing a traditional layered implementation of an *AUTOSAR* communication stack, which is on the one hand done manually by domain experts, and on the other hand via static analysis at source code level.

4.1 Component Based Middleware

Component based software engineering has found broad acceptance within software industry over the last years. However, the demand for cost-effective high-quality software forced component middleware to become reusable and configurable itself. As *AUTOSAR* relies on CBSE to gain cheaper and better applications, it sounds reasonable to apply the component paradigm not only at application level, but also to the middleware itself. Component based middleware promises to be more flexible than conventional middleware, when it comes to reuse, configuration, and application specific optimization.

When building component based middleware for *AUTOSAR*, two things have to be taken into consideration:

- Middleware components cannot rely on component middleware. Therefore, the *AUTOSAR* component model cannot be applied at *Basic Software* level.

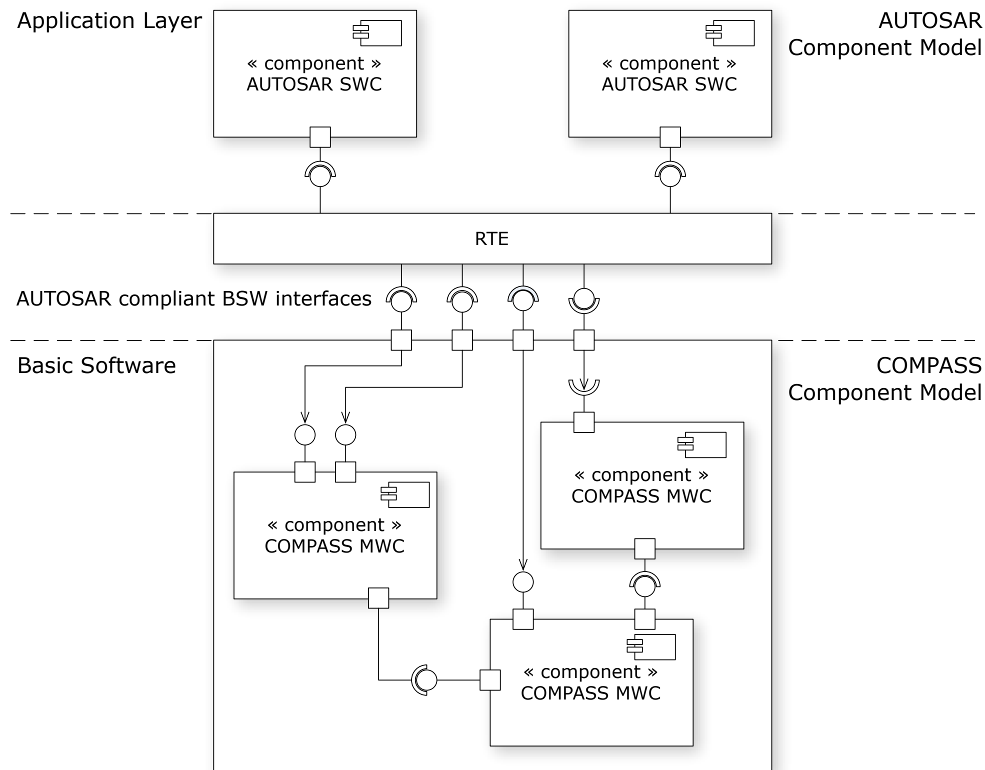


Figure 4.1: Component based middleware for *AUTOSAR*

- Middleware components have to provide standardized *AUTOSAR* functionality. Hence, the component model for middleware components has to be designed in line with the *AUTOSAR* standard—especially the standard's type system—to allow a seamless integration of component based middleware into *AUTOSAR* environments.

The following sections provide the cornerstones of *COMPASS* middleware for *AUTOSAR*, a component based communication middleware, developed within the scope of the research project *COMPASS* [COM07].

Figure 4.1 outlines the basic idea of component based middleware for *AUTOSAR*: Application components, labeled as *AUTOSAR SWC*, are located at the *AUTOSAR* application layer. They interact via the *Virtual Function Bus*, and are physically connected to the *Run-Time Environment*, as described in Section 3.4.2. The RTE exposes all middleware functionality to the application components, and adapts the user-defined application-

component-interfaces to the generic *Basic Software* interfaces. As consequence, the RTE has to be automatically generated during the application's build-phase. Neither the application components, nor the RTE, nor the standardized interfaces of the *Basic Software* are modified by the *COMPASS* approach. However, the component based middleware architecture at *Basic Software* level, as much as a component model for middleware components, is part of this thesis's contribution, and hence is described in detail within the following sections.

4.2 Component Model

Within the following section, the *COMPASS* component model for *AUTOSAR Basic Software* is specified by providing a component definition, and by defining rules for composition and interaction.

The specification of the component model is mainly provided by UML 2.0 meta-models. These meta-models formally define the vocabulary, required to describe *COMPASS* model elements and their dependencies. When instantiating a meta-model element[1], the resulting element is a *COMPASS* model element. When instantiated in turn, this model element finally leads to a specific implementation. To give an example, a meta-model element *ComponentType* can be used to instantiate a model element of type *Component*. Finally, a *COMPASS* compliant component architecture may contain a specific instance (implementation) of this model element (e.g., the *ErrorLogger* component, implementing centralized error logging). Abstract elements within the meta-model, (stereotyped *«abstract»*), must not be instantiated to gain model elements. These abstract meta-classes are mostly used to provide detailed information on aspects of instantiatable meta-classes.

4.2.1 Data Types

Data types are of great importance for describing components as much as component interaction. Therefore, a model describing data types has to be specified.

The *COMPASS* component model considers two levels of abstraction when defining data types, reflecting *AUTOSAR*'s type concept[2]:

[1] Classes in meta-models represent building blocks of models. Therefore, instantiating a class within a meta-model may again produce a class, but at a lower level (model level).

[2] The *AUTOSAR* standard defines three levels of abstraction: (i) the *Data Semantics Level*, (ii) the *Data Structure Level* and (iii) the *Data Implementation Level*.

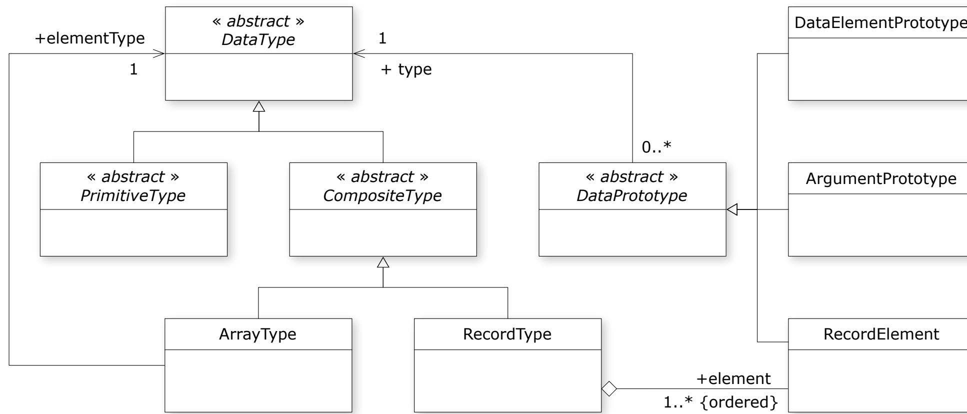


Figure 4.2: Meta-model of COMPASS data types (simplified)

1. **Structural Level**: At the structural level, common interface definition languages typically specify their data types by combining predefined primitive data types, to form various data structures, the user-defined types.

2. **Implementation Level**: At implementation level the mapping of data types and data structures to bits and bytes is of primary concern.

In heterogeneous component interaction both levels of abstraction have to be taken into account. ECUs of different type might encode data in different ways, so data types that match at structural level could mismatch at implementation level. Therefore, *COMPASS* provides rules for data conversion, which are based on the type meta-model provided in Figure 4.2. This model is equivalent to the one defined by the *AUTOSAR* standard [AUT08c]. In fact, *COMPASS* applies the component paradigm to *AUTOSAR Basic Software*, thus inventing a new—perhaps incompatible—type model is deliberately avoided.

Within the depicted model, class *DataType* is an abstract generalization of all data types of the *COMPASS* component model. The abstract classes *PrimitiveType* and *CompositeType* are both derived from *DataType* whereas *PrimitiveType* is the abstract superclass for all primitive data types. These are not shown in Figure 4.2 for simplicity but are named here:

- ***IntegerType*:** represents all integer types within the *COMPASS* component model (int, word, etc.)

- ***FloatType*:** represents all floating point types within the *COMPASS* component model (double, float, etc.)
- ***BooleanType*:** represents all boolean types within the *COMPASS* component model (bool, etc.)
- ***CharType*:** represents all character types within the *COMPASS* component model (char, TCHAR, unichar, etc.)
- ***StringType*:** represents all string types within the *COMPASS* component model (string, chararray, etc.)

All of them are concrete classes and hence can be instantiated to generate *COMPASS* data types.

In addition, the *COMPASS* type meta-model contains two classes for composite data types, represented by their abstract superclass *CompositeType*: *ArrayType* and *RecordType*. As depicted, an *ArrayType* is associated with exactly one *DataType* for all of its elements, while a *RecordType* contains an ordered set of *RecordElements*. *RecordElements* are concrete specializations of the *DataPrototype* class, which is associated with an arbitrary data type via the generalized *DataType* class.

The *ArgumentPrototype* and the *DataElementPrototype* are concrete specializations of *DataPrototype*, and are used within interface definitions. The *ArgumentPrototype* is used to specify arguments of operations within client-server interfaces, while the *DataElementPrototype* is used to specify data fields of shared-memory interfaces.

4.2.2 Component Definition

In Section 2.1, a general component definition was provided that comes to use now: A *COMPASS* component is a trusted architectural element of execution that is described by its well-defined points of interaction, the ports and interfaces, and that adheres to the *COMPASS* component model's composition- and interaction-standard.

COMPASS components are used to build local (ECU specific) communication middleware. Hence they interact with local components only (e.g., other *COMPASS* components or the local operating system), in order to provide means of transparent distributed interaction for *AUTOSAR* application components.

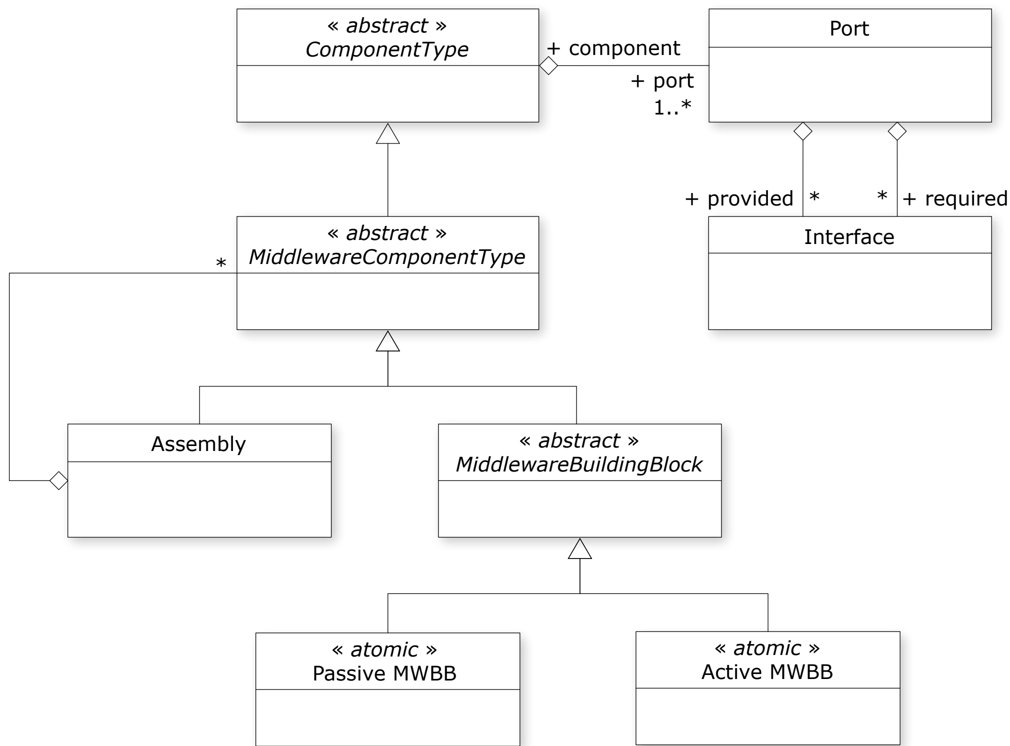


Figure 4.3: Meta-model of COMPASS components

4.2.2.1 Embodiment of COMPASS Components

Figure 4.3 provides a meta-model for *COMPASS* components, specifying how components may be represented by model elements within architectural models. *COMPASS* defines a hierarchical component model; hence, two general kinds of middleware components exist: middleware building blocks and assemblies.

1. ***Middleware Building Blocks*:** The most basic components of the *COMPASS* middleware are Middleware Building Blocks (MWBBs) that are derived from the abstract meta-model class *MiddlewareComponentType*. The class *MiddlewareBuildingBlock* itself is an abstract class that is implemented by two subclasses, *Passive MWBB* and *Active MWBB*. Note that both classes are stereotyped *«atomic»*. Middleware building blocks are atomic in terms of deployment (connected MWBBs have to be deployed within the same address space) and in terms of execution (operations of middleware building blocks must not be suspended by a

scheduler). Since atomicity always has to hold under both aspects, one single stereotype is used to describe them.

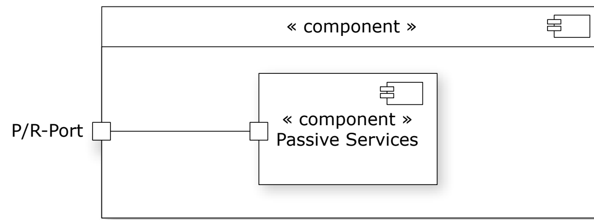
The *COMPASS* component model distinguishes two kinds of middleware components. In consequence, two kinds of middleware building blocks exist as mentioned above:

(a) **Passive Middleware Building Block**: Components of this type are executed within the client's thread of execution, which is requesting their service. They do not provide infrastructural interfaces for independent execution by a scheduler, but nevertheless may require external infrastructural services, like time service or logging facilities. Figure 4.4a shows the composite structure of a passive middleware component that comprises all its provided and required interfaces within ordinary provided- and required-ports.

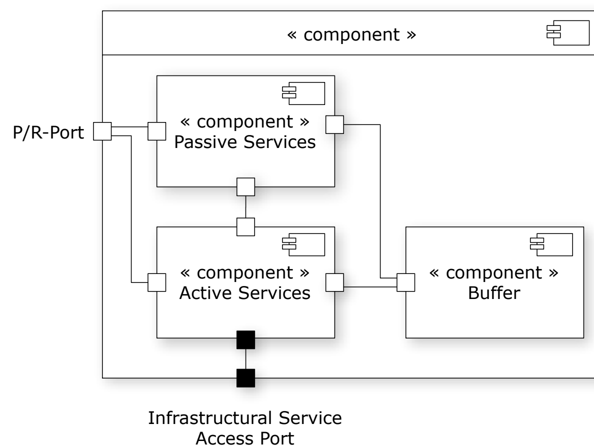
(b) **Active Middleware Building Block**: In contrast, active middleware components own at least one thread of execution that is independent from the client's one. Thus, they have to provide at least one interface for access by infrastructural services, like operating system schedulers or interrupt service routines (ISRs). Ports containing these interfaces are called *Infrastructural Service Access Ports*, and are denoted by a filled square. In Figure 4.4b one sub-component containing services that are executed within the middleware component's own thread of execution is depicted by the part labeled *Active Services.* It is an active sub-component and hence provides an *Infrastructural Service Access Port.* In addition the part requires and also provides passive services via ordinary provided- and required-ports. Active components may provide services that operate asynchronously to a client's thread of execution. Therefore, the client has to provide a call-back interface for reception of results, or may pull results on its own.

    As shown in Figure 4.4, component parts may expose their ports not only to other parts within the same component, but also to other middleware components, by connecting them to delegation ports of their enclosing middleware component.

Middleware components may also be stereotyped «singleton» indicating that the specific primitive must not exist more then once within a component architecture on one single ECU.



(a) Passive middleware building block



(b) Active middleware building block

Figure 4.4: Middleware building blocks

2. **Assemblies:** The *COMPASS* component model is a hierarchical model. Composed structures may be treated like basic building blocks, if viewed as black box. These composed structures are called assemblies and are represented by the class *Assembly* in the *COMPASS* metamodel. An assembly may contain composed middleware building blocks and other assemblies in a recursive manner. As an assembly represents some kind of frame around its interior building blocks, it provides no own middleware component characteristic (active or passive) but inherits one from its interior constituents:

    - An assembly is passive iff it contains only passive building blocks.
    - If the assembly contains at least one active building block (either another assembly or a middleware component), it is also an active one as it has to expose the *Infrastructural Service Access Ports* of

its inner active building blocks.

An assembly also contains information on the connections between its internal components. These model elements—the connectors—are covered in more detail in Chapter 5 and are omitted within the meta-model here for reasons of simplicity.

4.2.2.2 Component Type

To reuse and replace existing components within component architectures, components have to be classified in accordance to their design and abilities. Therefore, a component's type (also referred to as the component class) is determined by the set of well-defined points of interaction, especially by the component's interfaces and ports (see Section 2.1.1.2 and Section 2.1.1.3).

As described in Figure 4.3, middleware component types—assemblies and middleware building blocks—are subtypes of the abstract superclass *ComponentType* that specifies the look-alike of *COMPASS* components: A *COMPASS* component may own an arbitrary number of ports. In contrast to the *AUTOSAR* component definition that associates one port with exactly one interface (see Section 3.4), *COMPASS* ports may expose multiple interfaces of different types.

The *COMPASS* component model specifies two classes of interfaces[3] (see Figure 4.5):

1. **Client-Server Interface**: The client-server interface type contains an ordered set of *OperationPrototypes*, which contain another ordered set of *ArgumentPrototypes* each. According to this specification, a client-server interface consists of an ordered set of operations that are identified by their signature.

2. **Shared-Memory Interface**: The shared-memory interface type utilizes the *DataElementPrototype* data type (see Figure 4.2). An instance of a shared-memory interface is specified by the data structure written or read. Shared-memory interfaces are considered to be provided-interfaces of those components that own the shared-memory location, while they are considered to be required-interfaces for all other components that access extrinsic memory locations.

[3]This limitation was introduced, as *COMPASS* components are atomic in terms of deployment. Hence, interaction between middleware components may occur only via local procedure calls and shared memory access.

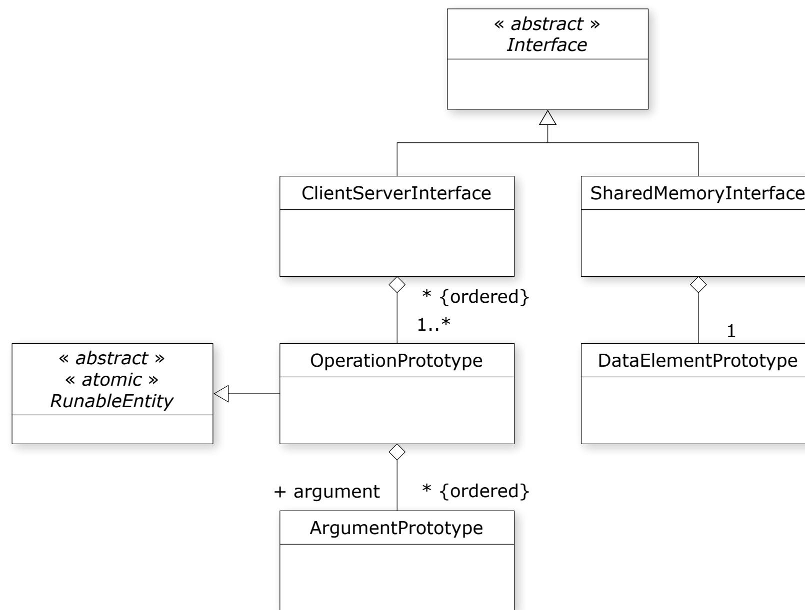


Figure 4.5: Meta-model of component interfaces

Two *COMPASS* components are type-equivalent, iff both expose type-equivalent interfaces and ports. Two ports are type-equivalent, if they comprise the same set of interfaces. Two interfaces are considered to be type-equivalent, if they contain the same set of operations and data elements, and are either both required-interfaces or both provided-interfaces. Two components that in total expose the same set of interfaces are not type-equivalent if the interfaces are partitioned in different ways, so the interfaces are exposed via different ports.

When it comes to sub-typing, a specialized component may replace a component of its super-type, if its interfaces contain at least the same set of operations and data types. In addition, all ports comprised in the super-type have to be comprised by the sub-type.

Any implementation of a specific component type may have its own, distinct name. Any instance of that implementation is considered to be an instance of the component's type.

4.2.3 Composition and Interaction

To build component architectures, every component model has to provide rules on how its components have to be assembled, and how they interact at run-time. Middleware components are artifacts that exist at a low level view of the overall system. Hence, the rules for composition and interaction prescribed by the *COMPASS* component model are rather simple but mandatory.

1. A connection between two middleware components is valid, iff
   - the connected interfaces are both client-server interfaces, or are both shared-memory interfaces.
   - the connected interfaces contain the same set of operations and data elements.
   - one is a provided-interface, and the other is a required-interface.
2. Connected *COMPASS* middleware components realize *AUTOSAR* communication middleware for a specific system node. They have to be deployed on the same ECU, and within the same address space. As a result, middleware components and assemblies are connected by local client-server connectors (local procedure calls), and by shared-memory connectors (direct memory access).
3. Interaction at middleware level is implemented as blocking invocation of operations (local procedure calls), or shared-memory access. Operations implemented by middleware components are atomic; they may not be suspended by a scheduler. This fact is expressed within Figure 4.5, where *OperationPrototype* implements the atomic *RunableEntity*. If an operation has to be interruptible, the ISR's developer has to ensure data integrity (e.g., for ECU registers or shared memory cells).
4. *COMPASS* middleware components are statically bound at compile time. Instantiation and initialization at run-time has to be performed at application start-up by one dedicated start-up-component.

Contrary to *AUTOSAR* software components, no middleware is available for *COMPASS* components, as *COMPASS* components are used to build component middleware. Therefore, all infrastructural services (operating system abstraction, scheduler, etc.) have to be encapsulated within middleware components that can be connected via infrastructural service access ports as described in Section 4.2.2.

4.3 Middleware Components

To make component based *AUTOSAR* compliant middleware available (and profitable) for industrial use, a component architecture for *Basic Software* based on the *COMPASS* component model has to be defined. The aspired benefit of increased reusability and adaptability for *AUTOSAR* middleware can only be gained, if this component architecture, as much as the involved middleware component classes, are not only well-defined but even standardized. This section in consequence specifies the full set of middleware component classes necessary to build the communication subsystem of the *AUTOSAR Basic Software*. The here presented concept can be applied to the remaining parts of the *Basic Software* in the same way, if desired.

To identify the middleware components of the communication subsystem[4], two distinct approaches are taken:

- In the first approach component classes are identified by human expertise. Domain experts manually classify the set of operations, prescribed by the *AUTOSAR* standard in terms of semantic relations. All operations and data structures related to each other are assigned to the same functional class, a component class [2]. The main issue is to decide if and how operations are related.
- The second approach is based on static source code analysis of an industrial reference implementation. This approach leads not only to one possible solution for decomposition, but to a full set of feasible ones.

4.3.1 Manual Component Classification

To manually classify middleware components for the *Basic Software*, a two phase procedure is defined:

1. **Identification Phase:** In the first phase, all operations, defined within the AUTOSAR specification of the *FlexRay Driver* layer [AUT06a, AUT08d], the *FlexRay Interface* layer [AUT06b, AUT08e] and the *FlexRay Transceiver Driver* layer [AUT06c, AUT08f] have been categorized with respect to their relation and binding[5]. In this way,

[4]For this thesis the chosen communication subsystem is FlexRay [MHB+01]. The described approach however is also applicable to the other communication subsystems standardized by *AUTOSAR*.

[5]By the writing of this thesis *AUTOSAR* R3 was released. Software analysis and decomposition was conducted for version R2. However, the changes between those two

groups of operations are identified, which are tightly coupled, or cover related aspects. In addition, the coupling between distinct groups is reduced to a minimum. Finally, all operations within a group, in conjunction with associated data elements, are considered to shape one specific middleware component.

2. **Optimization Phase:** In the second phase, the components formed in phase one are refined by deriving implementation variants. These variants are typically more specialized, e.g, they do not provide the full functionality of the base version, and therefore are less general, but significantly smaller in size and resource consumption. As a result, these variants of different size come to use when building custom-tailored and hence optimized middleware (see Chapter 5).

As result of the manual decomposition of the *FlexRay* communication stack the following *COMPASS* middleware component classes have been identified and specified:

1. **Base**: This component is the core of *COMPASS* middleware. It contains operations for node initialization, communication controller access, and data propagation. Thus, it is the hot-spot of FlexRay communication.

   The *Base* component is mandatory for each ECU, if the ECU participates in a distributed application. However, two versions have been specified: one that contains full functionality, and one that omits functionality of *cluster-communication*, and hence is smaller in size.

2. **Transmitter**: The *Transmitter* component contains all functionality to send data via the communication controller (CC). Again, two versions have been defined. The larger one is able to issue *TX confirmations* back to the *Base* component via a call-back interface, while the 'slimmer' one omits this feature.

3. **Receiver**: This component requires no real implementation. Data reception is performed by the CC that simply informs any recipient via a call-back mechanism. The *Receiver* component performs a simple call-delegation, but could, e.g., be extended for experimental purpose.

versions are rather small within relevant modules, and do not affect the overall outcome of this chapter. As a consequence, the presented approach as much as the results are also applicable to *AUTOSAR* R3.

4. **WUP**: The *WUP* component provides functionality of broadcasting a wake-up symbol on the bus. It is only required on ECUs containing an application that is capable of waking-up the entire automotive system from a low-power or suspend mode.

5. **Time Service**: This component provides operations to manipulate the timer-IRQ within the FlexRay CC. Multiple specialized variants of this component class have been created, and demonstrate the potential of the described approach. One for example is capable of manipulating absolute time only, while another is capable of handling global time in addition, and thus is larger in size.

6. **Status**: This component allows to read the CC status, and can be completely omitted for most applications.

7. **Media Test**: The *Media Test* component provides operations to send and receive a media test symbol to and from the bus. This functionality is seldom used, and thus provides great potential for middleware optimization.

8. **Transceiver Driver**: This component contains all operations that control the ECU's transceiver.

The total of all operations, contained within these components equals the *AUTOSAR FrIf* API. Therefore, it can be considered as a component based version of this *Basic Software* layer.

The full outcome of the applied two-phase procedure for manual component classification is documented in Appendix C. The appendix provides the complete information regarding the classification of all operations within the *FlexRay Driver* layer, the *FlexRay Interface* layer and the *FlexRay Transceiver Driver* layer, as much as their mapping onto identified middleware component classes.

4.3.2 Component Recognition by Static Analysis

Contrary to the manual identification of *Basic Software* components by domain experts, the second approach aims at automatic component recognition within an existing *AUTOSAR* middleware implementation.

4.3.2.1 Basic Principle

Two important properties of software components are encapsulation and separation. A well designed component contains a set of semantically related operations and memory locations that in total provide specific functionality exposed by the component's interfaces. When trying to find those related operations and memory locations within a global set of functions and data structures contained within monolithic or coarse-grained layered software, the coupling between all functions and all data structures has to be examined.

For the algorithm defined within this thesis two types of coupling are of interest:

Coupling via Control-Flow: Control-flow refers to the path of execution of a program. Two distinct functions within a program are strongly coupled by control-flow, if at least one of them passes control over to the other one. This is typically done by invoking the other function via a function call.

Coupling via Data-Flow: Data-flow refers to the flow of information during the execution of a program. As information is typically stored within specific locations like designated memory cells, coupling via data-flow can be observed by examining access to variables and structure fields. Two distinct functions are strongly coupled via data-flow if both access the same memory location, no matter if the type of access is read or write.

When taking these two types of coupling into account, two rules have to hold when performing automated component recognition:

1. Operations within one component may be coupled (vertical coupling) either by control-flow or by data-flow.
2. Operations are (considered to be) not coupled, if they are contained within distinct components, even if they are linked by control-flow or data-flow (horizontal coupling). Operations linked in such way have to be exposed in a component's interface (a provided-interface of the component that contains the operation or owns the memory location, and a required-interface of the component that invokes the operation or accesses the memory location).

To decompose existing non-component based programs into a set of reusable, exchangeable and thus independent components, the proposed algorithm gathers information on horizontal and vertical coupling by performing a static analysis of the program's source code. Utilizing the analysis's results, all functions of the program are divided into sets of decoupled, independent components.

The analysis is performed at compile-time and is independent of any test cases. The recognized components are determined with respect to all possible program runs. The algorithm is flow-insensitive (independent of intra-procedural control-flow) and performs a single pass on the input program. It does not consider the different calling contexts of functions and is therefore context-insensitive.

To determine the vertical and the horizontal coupling, the algorithm considers three categories of information, which are gathered by traversing the program's abstract syntax tree (AST):

Functions: The functions of the input program are the basic ingredients for component recognition, and in the end are assigned to one specific component class each, recognized by the algorithm's decomposition.

Function calls: Any function call means coupling by control-flow. Thus, function calls imply a semantic relation between the calling and the called function.

Structure field types and names: To gather information on coupling by data-flow, the proposed algorithm relies on type-information without the consideration of any concrete instances of data. As any function that uses a specific type has a semantic relation to all other functions also using this type, type based analysis turned out to be sufficient for the algorithm's purpose.

However, fundamental data types typically do not imply any semantic relations. In consequence, only user defined data types are considered during analysis. In detail, the collected type-usage information consists of structure types and field names, similar to the type-based alias-analysis *FieldTypeDecl* described in [DMM98].

4.3.2.2 Cohesion Analysis Algorithm

When developing an algorithm based on static analysis, various options regarding complexity and thus execution time exist. The described algorithm was developed with respect to scalability, hence its complexity is kept linear to the size of the analyzed source code. In addition, it incorporates configuration data, especially data on late bound function pointers and on domain specific properties, to provide linear complexity and to find sufficient solutions quickly.

The algorithm (*Cohesion Analysis*) defined on page 62 traverses a program's AST and collects information on call-dependencies between functions, and on type-based memory access within functions. This information is further on used to build a component graph that is finally used to classify middleware component classes.

The component recognition algorithm is denoted in pseudo-code. This code uses named sets and tuples to describe its mode of operation in an implementation-independent way. Hence, those named sets and tuples will be defined below, before providing the algorithm itself.

Definition 1 (**Call Graph**). *A call graph G_C of a program P is represented by a pair of sets N_C and E_C, where the graph's nodes $n \in N_C$ represent marked functions $fn \in F_P$ of a program P, and the edges $e \in E_C$ denote calls between these functions.*[6] *To keep the analysis focused on the program code itself, helper-functions are held off from the call-graph by considering relevant—manually marked—functions only.*

$$F_P = \{f \mid f \text{ is a function} \quad \wedge \quad f \in P\} \tag{4.1}$$

$$G_c = (N_C, E_C) \mid N_C = \{fn \mid fn \in F_P \quad \wedge \quad fn \text{ is marked }\}, \quad E_C = \{\{a, b\} \mid a, b \in N_C \quad \wedge \quad a \text{ calls } b \} \tag{4.2}$$

Definition 2 (**Type-Based Field Usage**). *A type-based field usage is a pair (s, d) where s denotes the structure type of the used data structure, and d denotes the field name—not its type—of the used data field. Again, relevant structure types and field names may be manually marked, to keep the analysis focused on specific program elements.*

[6] For the purpose of the *Cohesion Analysis* algorithm, edges are undirected. Hence, they are represented by a set of two (connected) nodes rather than by an (ordered) pair: $\{a, b\} = \{b, a\}$ but $(a, b) \neq (b, a)$.

$$U_T = \{(s,d) \mid s \text{ is a structure type } \in P \wedge d \text{ is field of } s \wedge s \text{ and } d \text{ are marked } \} \quad (4.3)$$

Definition 3 (**Type-based Field Usage Graph**). *A type-based field usage graph G_U is represented by a pair of sets N_U and E_U. The graph's nodes $n \in N_U$ represent a type specific abstraction of data usage—the type based field usage N'_{SF}—as well as marked functions that use specific data—N'_{Fn}. Edges $e \in E_U$ connect functions and type-based field usages, and thus represent usage of data by specific functions.*

$$\begin{aligned} G_U = (N_U, E_U) \mid\ & N_U = N'_{SF} \cup N'_{Fn} , \\ & N'_{SF} = \{n \mid n \text{ is marked } \wedge n \in U_T \} , \\ & N'_{Fn} = \{m \mid m \text{ is marked } \wedge m \in F_P \wedge \\ & \qquad \exists\, u \in U_T \,.\, u \text{ occurs in } m \} , \\ & E_U = \{\{a,b\} \mid a \in N'_{Fn} \wedge b \in N'_{SF} \wedge \\ & \qquad b \text{ occurs in } a \} \end{aligned} \quad (4.4)$$

Definition 4 (**Component Graph**). *A component graph G_P is denoted by a pair of sets N_P and E_P. Its nodes $n \in N_P$ represent the program's data usage abstractions, and the program's functions. Its edges represent calls between the program's functions, or usage of data by a function. The component graph is calculated by creating a set union of the program's call graph and its type-based field usage graph. It unites all gathered information on coupling via control-flow and on coupling via data usage.*

$$G_P = G_C \cup G_U = (N_P, E_P) \mid \quad N_P = N_C \cup N_U , \quad E_P = E_C \cup E_U \quad (4.5)$$

The component graph contains disjoint sub-graphs—*connected components* in a graph-theoretic sense—that represent coupled, self-contained functionality, and thus serve as candidates for components in the software engineering sense.

With the provided definitions in mind, the algorithm specified in Algorithm 1 has to adhere to the following work-flow:

1. **Annotate function pointers.** Any function pointer within the input program P has to be associated with exactly those function(s) it will point to at run-time. Within the analyzed *AUTOSAR Basic Software*, function pointers are often used to implement configuration-time late binding. The address of a function pointed to by the function pointer is manually set to the address of an available function after compile-time as part of the system's configuration. As this address is not known at compile-time, and thus is not available at analysis-time, manual annotation with points-to information, or static assignments of the addresses of configured functions to the respective function pointers at source-code level, is mandatory for the algorithm to calculate valid results.

2. **Mark relevant functions.** Most software utilizes common helper functions or compiler built-ins for e.g. memory manipulation. To avoid assigning these functions to program specific components, all functions of interest (within this thesis also referred to as relevant functions) have to be marked by a domain expert. In practice, the set of irrelevant functions is smaller than the set of relevant ones. Therefore, the algorithm takes a set of irrelevant functions as input value ($\overline{F_M}$). This set is later on used to calculate the set of all relevant (and thus marked) functions F_M (Algorithm 1, line 3).

3. **Mark characteristic structure fields.** The most important task of a domain expert within the proposed work-flow is the identification of relevant data structures respectively their relevant fields. Some structures are closely related to exposable component functionality, but other ones are used for internal purpose only. By taking only relevant fields into account, the algorithm is sensitive to dedicated component functionality only. Normally, the set of irrelevant structure fields is smaller than that of relevant ones. Consequently a set of fields that have to be ignored ($\overline{D_M}$) is passed to the algorithm, which calculates the set of all relevant and therefore marked type/field-name pairs D_M via set-subtraction of $\overline{D_M}$ from the set of all type/field-name pairs of P (S_P)(Algorithm 1, line 4).

All three steps so far imply manual work by domain experts. However, the gained result provides the input data, required by the *Cohesion Analysis* algorithm: the annotated source code of the program under analysis, a set of irrelevant functions, and a set of irrelevant structure fields. All following steps are part of the algorithm's implementation and are therefore executed automatically.

Algorithm 1: Cohesion Analysis

```
1  ComponentRecognition(P, ‾F_M, ‾D_M):   C_P
2  begin
3      F_M ← F_P − ‾F_M;     // determine relevant functions
4      D_M ← S_P − ‾D_M;    // determine relevant structure fields
5      N_C ← ∅;  E_C ← ∅;  N_U ← ∅;  E_U ← ∅;
       // iterate over all relevant functions in a program
6      foreach f ∈ F_M do
7          N_C ← N_C ∪ {f};
           // iterate over all expressions and subexpressions of function
8          foreach exp ∈ expressions(f) do
9              switch exp do
                   // collect structural data usages
10                 case lhs.rhs or lhs −> rhs     // dot or arrow expression
11                     s ← structure type in lhs;
12                     d ← structure field name of rhs;
13                     if (s, d) ∈ D_M then
14                         N_U ← N_U ∪ {f} ∪ {(s, d)};
15                         E_U ← E_U ∪ {{f, (s, d)}};
16                     end
17                 end
                   // collect call graph data
18                 case f_c(...)  // function-call-expression
19                     E_C ← E_C ∪ {{f, f_c}};
20                 end
21                 otherwise    // any-other-expression
22                     skip;    // no information is extracted
23                 end
24             end
25         end
26     end
       // build undirected component graph
27     G_P ← (N_U ∪ N_C, E_U ∪ E_C);
       // extract components via reachability
28     C_P ← ∅
29     while G_P ≠ (∅, ∅) do
30         n ← choose_node(G_P)      // choose some node n
31         G_S ← reachable_subgraph(n, G_P)
32         G_P ← G_P − G_S      // remove subgraph G_S from G_P
33         C_P ← C_P ∪ {nodeset(G_S)}
34     end
35 end
```

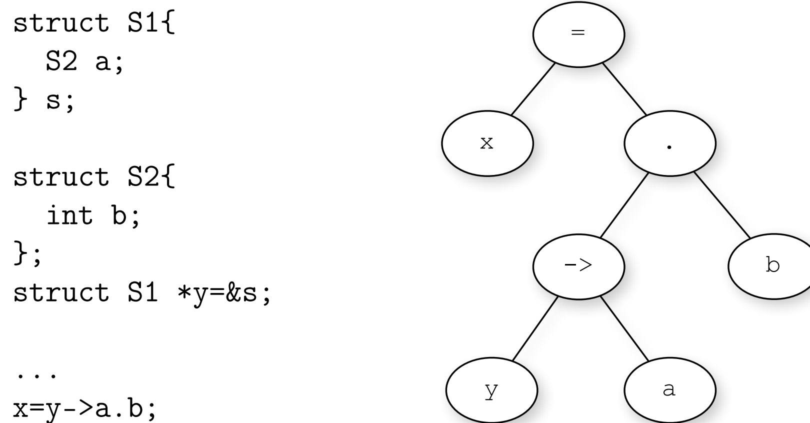


Figure 4.6: Source code example and AST

4. **Calculate the call graph.** A call graph, $G_C = (N_C, E_C)$, is computed from P's AST where functions of the program are represented by nodes, N_C, and calls are represented by edges, E_C, between the calling and the called function (Algorithm 1, line 18–20).

5. **Calculate usage graph.** A usage graph, $G_U = (N_U, E_U)$, is computed where functions and type-based field usages are represented by nodes, N_U, and field access within a specific function is represented by an edge, E_U, between the function node and the type-based field usage node. To identify field accesses, the occurrence of arrow- and dot-expressions within the AST is analyzed (Algorithm 1, line 10–17).

   Figure 4.6 provides a C-code snippet, and an AST snippet representing the last line of the code. The *Cohesion Analysis* traverses the program's AST and finds all occurrences of field accesses in user defined data structures. In the given example, two structure types (*S1* and *S2*) with one data field each (a and b) are defined. As a result two data field accesses ($(S1, a)$ and $(S2, b)$) are collected by storing the structure-type of the left-hand-side, and the field name of right-hand-side of the dot- and the arrow-operator.

6. **Calculate component graph.** A component graph, $G_P = (N_P, E_P)$, is calculated by creating a set union of the call graph and the usage graph. It unites all gathered information on coupling via control-flow and on coupling via structure type based data field usage (Algorithm 1, line 27).

7. **Extract components from component-graph.** The algorithm's

final step is the extraction of all disjoint, connected sub-graphs—the components—from the component graph. The algorithm performs the extraction via a reachability calculation. Its output C_P is a set of components, where each component is represented by a set of functions and structure fields contained within the component (Algorithm 1, line 28–34).

Finally, a domain expert may further group multiple of the automatically computed decoupled components into single components, if desired. This step becomes rather handy, if the analysis identifies many small components. Although large numbers of very small components imply high potential for optimization during automatic middleware synthesis, it is feasible to combine some into larger components for reasons of economy in maintenance and versioning.

Compared to the manually gained decomposition of Section 4.3.1, the *Cohesion Analysis* was able to recognize not only one, but a set of valid decompositions (see Section 6.1.1 for discussion). However, the one solution manually created by domain experts is contained within the set of solutions, calculated by the algorithm. Detailed results of the *Cohesion Analysis* applied to an industrial implementation of an $AUTOSAR$ communication stack can be found in Chapter 6.

The described algorithm inherently depends on configuration specific data, especially late bound function pointers that are assigned at configuration time and hence have to be annotated manually. Therefore, distinct configurations may lead to distinct results. For the subject of this thesis, one representative configuration was identified, and used. To get a more general decomposition without assuming one representative configuration, multiple analysis results of distinct configurations may be fused (see Section 6.3 on future work).

The *Cohesion Analysis* algorithm was designed to feature linear complexity and generally applicable results, to achieve good scalability and hence industrial exploitability. As a consequence, it requires manual annotations—marks for relevant functions and data fields—by domain experts. As any expert intervention is costly, those manual annotations are the drawback of the described algorithm. However, the amount of work for marking functions and data fields is nearly the same as the one for manual decomposition. Therefore, the analysis based approach clearly outnumbers the manual approach when it comes to reuse (e.g. in case of different configuration scenarios). Ongoing research in addition aims at a significant reduction of required annotations without loosing the nearly linear complexity of the used analysis.

4.4 Summary

This chapter applied the component paradigm to *AUTOSAR Basic Software*, in detail to its communication stack. Therefor, the *COMPASS* component model was defined. The *COMPASS* component model is fully compatible to *AUTOSAR Basic Software* in terms of external interfaces. Hence, component based *Basic Software* can seamlessly be integrated into existing (traditional) software systems.

In addition, middleware component classes were specified, which come to use in automatic middleware synthesis as described in Chapter 5. To identify middleware component classes that comply to the actual *AUTOSAR* standard, an *AUTOSAR* reference implementation of the communication subsystem has been decomposed. Decomposition was on the one hand done manually, and on the other hand via static analysis at source code level. This analysis, the *Cohesion Analysis*, calculates component classes, using information on control-flow dependencies and usage of abstract memory locations, which are identified via structure types and field names respectively. To gain linear complexity for the proposed analysis algorithm and to increase the result's quality, domain know-how in terms of manual annotations at source code level is utilized. Creating this annotations may cause nearly the same costs than manual decomposition of an existing implementation. However, a major benefit arises if the analysis is applied to different configuration scenarios (especially in presence of late bound function pointers).

Manual decomposition led to eight component classes of middleware components; the *Cohesion Analysis* identified not only one but a set of equivalent possible decompositions. However, the manually identified solution was also found by the static analysis, which clearly demonstrates its abilities. A more detailed description of the calculated analysis results is given in Chapter 6.

Chapter 5

Middleware Synthesis

In Chapter 4 the component paradigm was applied to *AUTOSAR Basic Software*. As a result, middleware components have been specified, which serve as basic building blocks for component based communication middleware. The proposed architecture fully resembles interfaces and functionality of conventional *AUTOSAR* middleware, but provides increased reusability and maintainability due to its component based nature. Based on the presented design, this chapter provides the second major contribution of this thesis. It describes a model driven methodology—the *COMPASS* methodology—that facilitates automatic synthesis of communication middleware from application models, deployment specifications, and prefabricated middleware components.

5.1 Methodology

Within the *AUTOSAR* methodology [AUT08b], an application's component architecture is described by platform independent models that on the one hand are subsequently used to configure generic component middleware, and that on the other hand are required to generate the application's Run-Time Environment (RTE)[1].

The *COMPASS* methodology is a model driven approach for component based application development. *COMPASS* introduces automatic generation of custom-tailored component based middleware to *AUTOSAR*'s workflow. Based on the MDA development phases outlined in Section 2.2.3, the

[1]The *RTE* is generated in line with the *AUTOSAR* methodology, and thus is not subject to this thesis. Nevertheless, *RTE* functionality that is related to communication and interaction is reflected within architectural middleware-patterns provided in Section 5.3.1 on a basic level.

COMPASS methodology can be described by the following consecutive activities[2]:

1. **Specification of CIM and Platform Model**: As the application under development targets the *AUTOSAR* domain, CIM and platform model are defined within the *AUTOSAR* context. However, to satisfy the requirements of *COMPASS*, they have to be extended by concepts of the *COMPASS* component model:

   - The *AUTOSAR* compliant CIM is extended by *COMPASS* specific vocabulary, e.g. by the term 'middleware component' and the concept of component based middleware.
   - The platform model has to contain at least a description of all system parts and their configuration (e.g., ECU types, hardware resources, interconnecting bus-systems), a specification of available software interfaces, and a middleware specification. Within a *COMPASS* compliant platform model this middleware specification on the one hand includes the set of available interaction specific architectural patterns—the connector templates as described in Section 5.3.1—and on the other hand contains specifications of all available middleware components as proposed in Section 4.3.

2. **Specification of PIM**: Applications are designed by specifying high-level component architectures in a platform- and deployment-independent view, using UML 2 component diagrams. Platform independent application models contain application components only, which are connected by explicit connectors (see Section 5.2.1) in a location transparent way. Explicit connectors are stereotyped by specific communication styles (e.g., sender-receiver, client-server). They may be marked with communication specific attributes and QoS attributes to guide the PIM-to-PSM transformation.

   To support interaction specific annotations required by the *COMPASS* methodology, a UML 2 profile can be used [RMRR98] to increase the application models' expressiveness and readability.

3. **Deployment Specification**: The application components' deployment, and all relevant physical parts of the electronic system in terms

[2] Appendix B lists all UML 2.0 diagram types that are used within each phase of the development process.

of nodes and buses are described in a deployment model. Communication specific requirements on the component middleware—the characteristics of communication channels between connected components—strongly depend on the components' physical location. Hence, the subsequent PIM-to-PSM transformation extracts and utilizes this characteristics from the deployment model, to select appropriate middleware patterns.

4. **Transformation of PIM to PSM**: The core of any MDA compliant development process is the transformation of PIMs to PSMs. Within the *COMPASS* approach, the platform independent application model—the component architecture at application level—is transformed into platform specific component architectures for each system node.

   The *Connector Transformation*, as described in Section 5.3, is accomplished by injecting platform- and distribution-specific component architectures in place of connector artifacts within the PIM. The PIM in consequence is commuted to a PSM. The injected component architectures consist of prefabricated middleware components (see Chapter 4), which are assembled in accordance to connector templates. Additionally, interface adapters as part of the *RTE* are generated. These artifacts adapt user-defined interfaces of application components to the generic interfaces of the *AUTOSAR* compliant component middleware. The generated platform specific component architectures later on are transformed into ECU specific executables, system configuration data, and control files by subsequent model-to-code transformations.

5. **Transformation of PSM to Code**: The platform specific component architectures, generated by the PIM-to-PSM transformation, are transformed into machine executable code. All application component binaries, all middleware component binaries, and all remaining binaries, for example parts of the *RTE* like protocol handlers (see Section 5.3.1), are bound according to generated control files. Therefore, one machine executable, which contains application- and middleware logic, is generated for each system node that is part of the application.

6. **Deployment and Configuration**: To finally gain a running system, all generated binaries have to be deployed on their target nodes. Besides that, the generated system configuration has to be applied, to assure a valid system state.

5.2 COMPASS Application Models

COMPASS application models are based on platform independent *AUTOSAR* models. Thereby, *AUTOSAR* models are augmented by interaction specific attributes as much as by model artifacts for explicit connectors. The following sections introduce explicit connectors as first-class architectural elements, and in addition adds various types of contracts at model level to hold supplemental attributes of model artifacts.

5.2.1 Explicit Connectors

In well established component models like OMG's CORBA Component Model (CCM) [OMG06], or Microsoft's Component Object Model (COM) [Mic09], but also in *AUTOSAR*, all functionality related to component interaction and communication is implemented by component middleware. Matters of distribution or heterogeneity are completely transparent for the components. Connectors as defined within these component models are model level entities denoting the relation between associated component interfaces. Since they provide no implementation on their own, but represent interaction on an abstract level only, these connectors are referred to as implicit[3] connectors within this thesis.

However, connectors represent middleware functionality, hence they can be associated with the implementation of specific middleware parts. Consequently, it is feasible to grant them first-class status within a component model. In contrast to implicit connectors that provide an abstraction of a system's middleware, these connectors resemble parts of the middleware. They make the system's middleware explicit, and thus are referred to as explicit connectors.

Explicit connectors describe idiomatic patterns of functionality, provide mechanisms of data- and control-flow [Kel07], and are associated with different architectural styles [SG96, Sha93]. Within application models, explicit connectors express interaction specific attributes, and provide additional fine grained information on communication and interaction specific properties.

The implementation of explicit connectors may become rather complex, depending on communication style and deployment scenario. Thus, explicit connectors may also be constructed in a component based way by assembling middleware components like the ones defined in Section 4.2.

[3] Connectors of this type implicitly denote component middleware that is transparent for the components.

Although explicit connectors resemble components within the proposed approach, they differ in many aspects. Unlike components, a connector changes its materialization during its life-cycle due to model transformations:

1. In platform independent models the explicit connector is an abstract representation of component interconnection, specifying interaction and communication specific properties.

2. In platform specific models an abstract explicit connector is transformed into a concrete set of middleware components. This middleware component architecture implements the explicit connector's interaction behavior and communication paradigm. However, after successful transformation, all connectors within the PSMs are implicit local connectors only.

3. At deployment- and finally at run-time, explicit connectors are no longer visible. True components representing the explicit connectors' functionality are deployed and executed.

Figure 5.1 depicts the PIM- and the PSM-phase within an explicit connector's life-cycle as described above. For demonstration purpose, the depicted example is specified as distributed application that is deployed onto two system nodes, *Node 1* and *Node 2*

- The platform independent model shows two application components, A and B. B requires services, provided by component A via the interface IA. The components' associated interfaces are connected according to Section 2.1.2. The connector, depicted as usage-relation, constitutes an abstract entity, and allows the annotation of communication and interaction specific attributes.

  This PIM so far is in line with conventional *AUTOSAR* application models. However, the PIM's semantic is augmented for the purpose of this thesis: The connector C between B and A is considered to be an explicit connector, and thus is granted first-class status within the model. As a result, C can be transformed into a platform specific middleware component architecture, contained within a PSM.

- The platform specific model shows the explicit connector C after its transformation: The PSM consists of two sub-models, one for each system node. The abstract model element of C from the PIM has been transformed into two components, CF_1 and CF_2. These two components in total provide the implementation of the explicit connector C,

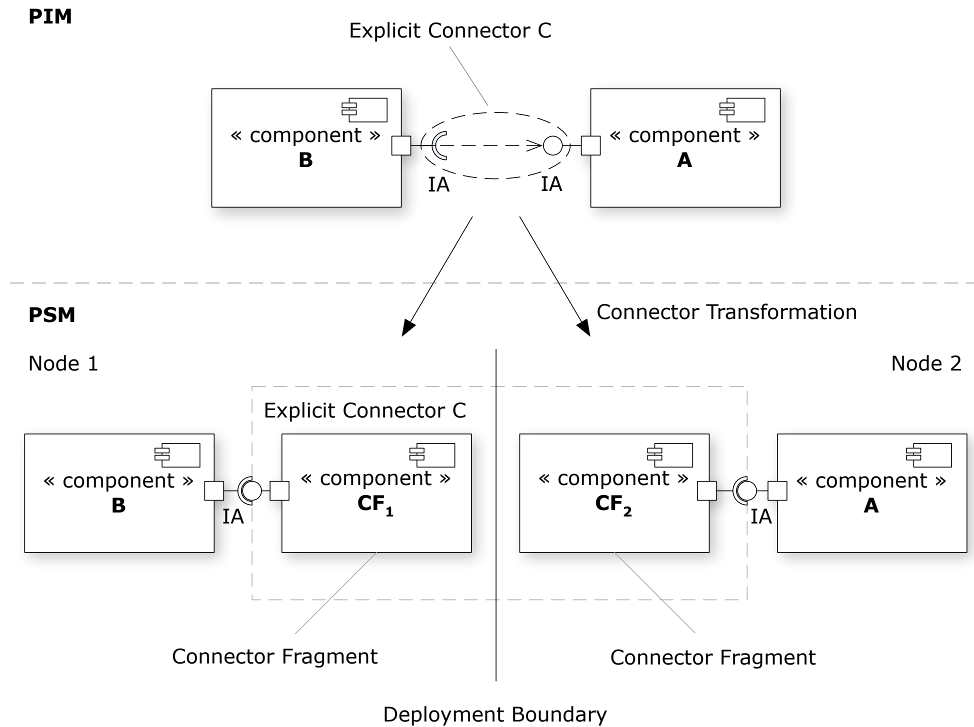


Figure 5.1: PIM to PSM transformation of connector artifacts

where each of them provides a part of the overall functionality. Therefore, this specific type of component is called connector fragment.

The connector fragments offer the same interfaces that were connected by C within the PIM, here IA. The application components B and A are directly connected to the fragments via an implicit connector. All connector artifacts within PSMs are implicit connectors, typically local procedure calls or shared memory connectors. To emphasize this fact within this thesis, the ball-and-socket notation is used for connectors in PSMs, while the usage-relation notation is used for explicit connectors in PIMs.

5.2.1.1 Deployment Anomaly

Considering the example given in Figure 5.1, an additional characteristic of explicit connectors in distributed systems becomes visible: The explicit connector C resides between the connected application components A and B. If A and B are deployed on the same single node, the connector's implementation

has to be deployed there, too. However, if the two application components are deployed on different nodes, as assumed in the given example, it is no longer trivial to deploy the connector. In fact, the connector artifact within the PIM has to be split into two pieces within the PSMs (the sub-models for each node). These connector fragments subsequently have to be deployed along with their associated application components. CF_1 has to be deployed on the same node as B while CF_2 has to be co-located with A. As a result, the functionality of connector C crosses the deployment boundary.

This characteristic is referred to as deployment anomaly in literature [Bál02, BP01]: A connector as one single model element may own an implementation, which is spread over multiple system nodes.

The representation of an explicit connector within a PSM is a set of connector fragments that are full-fledged, typically prefabricated, middleware components. Connector fragments contain functionality and implementation of the system's component middleware. Remote interaction and communication is handled by these fragments in a black-boxed way, and hence is not of concern for the application components B and A. Nevertheless, all communication- and interaction-specific properties can be specified within an application model via properties of the connector C.

As denoted in Table 5.1, a connector between components within the same address space is called Intra Process Connector (IPC), one between different address spaces but on the same Electronic Control Unit (ECU) is called Extra Process Connector (XPC) and one between different ECUs, which is obviously also one between different address spaces, is called Extra Node Connector (XNC).

Abbr.	**Name**	**Interaction**	**ECU**	**Addr. Space**
IPC	Intra Process	local	co-located	same
XPC	Extra Process	inter process	co-located	different
XNC	Extra Node	remote	distributed	different

Table 5.1: Connector classification by interaction type

All three types of interaction may occur within an *AUTOSAR* compliant application, depending on the application's deployment scenario. Hence, the middleware synthesis described within this chapter inter alia determines application requirements in terms of interaction types. As a result, only middleware components that implement those required interaction types are incorporated into custom-tailored communication middleware.

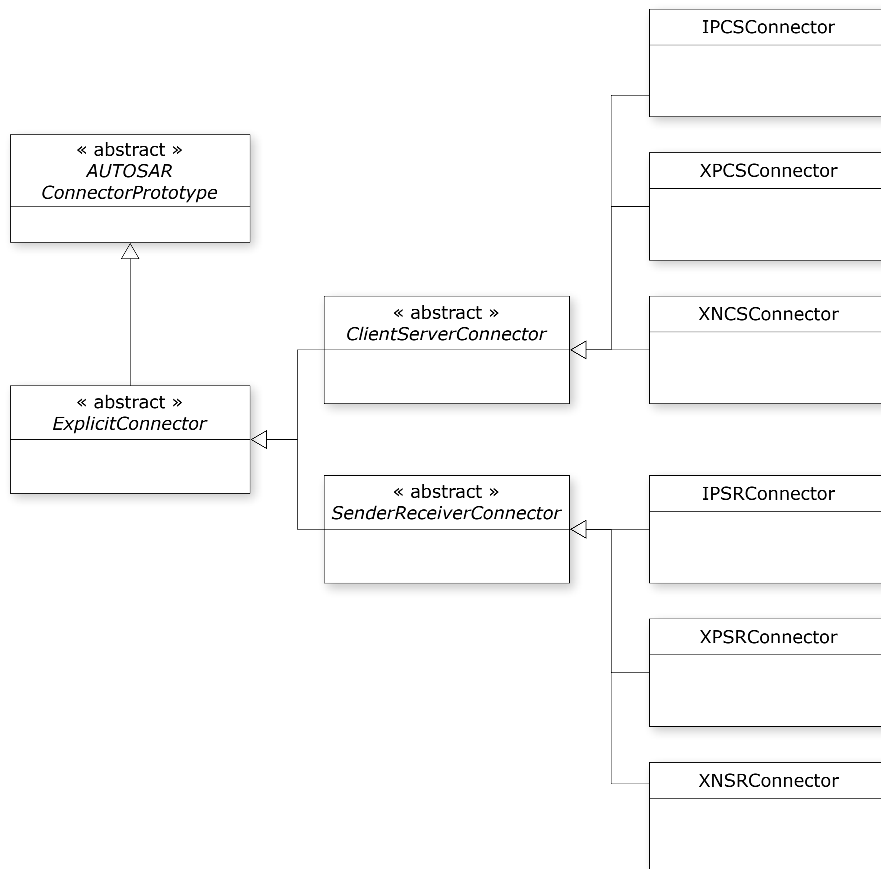


Figure 5.2: Meta-model of *COMPASS* explicit connectors

5.2.1.2 Interaction Styles

In accordance to the prevailing specification of *AUTOSAR*, any interaction within an application is mapped onto two basic interaction styles: the client-server paradigm and the sender-receiver paradigm, which have to be implemented by *AUTOSAR* middleware.

Figure 5.2 depicts a meta-model for explicit connectors in platform independent *COMPASS* application models. All specialized explicit connectors are derived from the common superclass *ConnectorPrototype*, which is *AUTOSAR*'s connector base class. The subclasses are characterized by two dimensions that classify an explicit connector at application level:

- **Interaction Paradigm:** The first dimension represents the connector's interaction style. Hence, the abstract superclass *ExplicitConnec-*

tor is specialized by two also abstract classes, *ClientServerConnector* and *SenderReceiverConnector*.

- **Component Deployment:** The second dimension represents the interaction type with respect to the deployment scenario of the connected application components (see Table 5.1). Two connected components may be deployed within the same address space (IP), on the same ECU, but within different address spaces (XP), or on different ECUs (XN). Note these prefixes in Figure 5.2.

As a consequence, the implementation of six types of explicit connectors has to be supported by *COMPASS* communication middleware to be in line with *AUTOSAR*.

5.2.2 Contracts

To strengthen the reliability and predictability of component based applications, guarantees about properties and behaviors of application elements are formalized in contracts [Mey92a, Mey92b, Mey03, RS02, CHJK02, NB02]. Contracts specify requirements and provisions of associated elements. In general a contract consists of two obligations:

1. The client, who is requiring functionality from another element, has to satisfy the preconditions of the provider.
2. The provider, who is the supplier of the required functionality, has to fulfill its post-condition, if the client's precondition is met.

When modeling a component architecture, contracts are denoted as model artifacts, and are associated with the model element they refer to. The model depicted in Figure 5.3 revisits the example from Section 5.2.1, but is enriched by twelve contracts, the artifacts stereotyped «contract». These contracts provide important constraints for automated model transformation, especially for automatic middleware synthesis. In addition they may also be used for model level system validation and verification.

For the process of automatic middleware synthesis described within this thesis, five main types of contracts are used[4]:

[4]Of course more types exist, but are not of relevance for the proposed methodology.

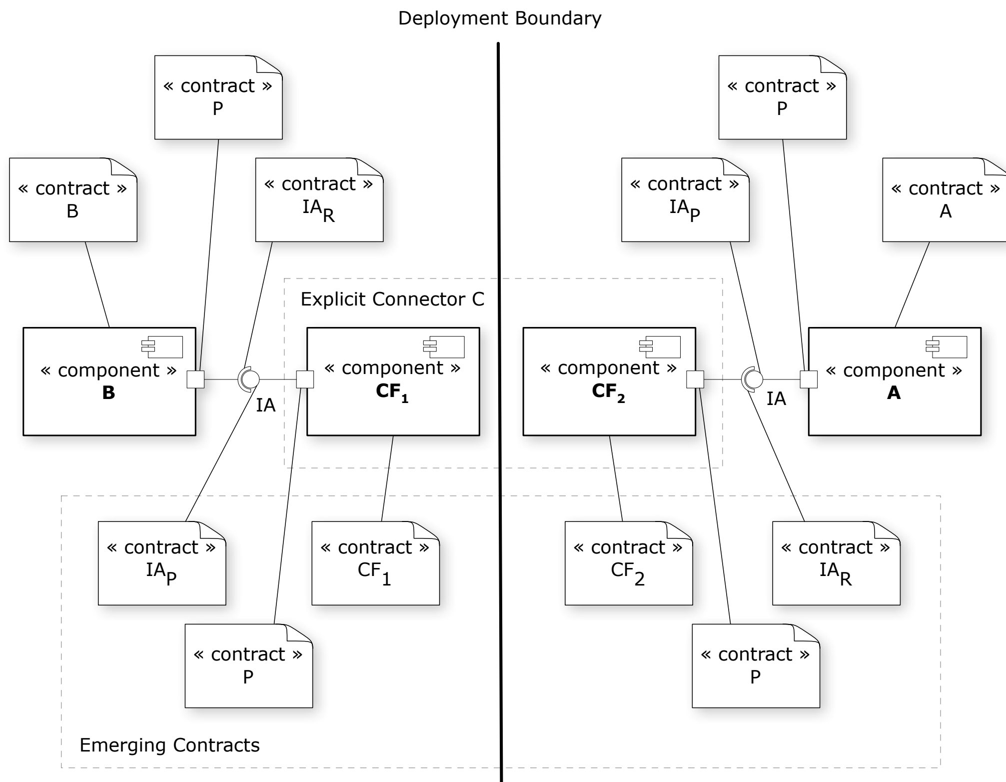


Figure 5.3: Notation of contracts in UML

1. **Component contracts** are associated with component artifacts. Typical component contracts deal with resource requirements or deployment restrictions, like a component's memory footprint, required system resources, or the required ECU type. In Figure 5.3, the contracts B, A, CF_1, and CF_2 are component contracts.

2. **Interface contracts** specify requirements and provisions for composition and interaction. Attributes within these contracts typically describe services and properties of a component's interfaces, like operation signatures, accessible data elements, or temporal properties like worst-case execution time (WCET) for operations. The contracts labeled IA_P and IA_R are typical interface contracts representing augmented interface definitions. If an interface contract contains an interface specification in terms of operation signatures and data element definitions only, the contract artifact may be omitted, as this information is already contained within the standard UML interface definitions.

3. **Port contracts** are associated with ports and their interfaces, and deal with the relation between them, like execution- and message-sequences, or the timing between port invocations. Behavior protocols like described in [PVB99] are typically contained within port contracts. Port contracts in Figure 5.3 are the ones labeled with *P*.

4. **Connector contracts** are associated with connectors at application level and contain constraints related to communication channels like worst-case propagation time or the required communication style [BP04]. Connector contracts are typically attached to explicit connectors at application level. As Figure 5.3 depicts a PSM after transformation, it contains no explicit connectors and no connector contract.

5. **Platform contracts** specify properties of platform elements like ECUs or bus systems (ECU type, available random access memory (RAM), available read-only memory (ROM), bus timing, etc.). Platform contracts are related to system nodes and hardware resources, and thus are typically denoted within deployment specifications, e.g., within deployment diagrams.

5.2.2.1 Emerging Contracts

Contracts associated with software components (application components) of a *COMPASS* application have to be provided by the components' manufacturer. When applying the *Connector Transformation* as described in Section 5.3, a wide set of additional contracts emerges within transformed PSMs, depending on composition, deployment and interaction. These additional contracts originate from prefabricated platform specific middleware components that are injected into the PSMs during model transformation. Emerging contracts provide supplemental fine-grained information on the synthesized component architecture, and thus facilitate model-level validation of the overall software system under development.

Figure 5.4 shows the process of middleware component development from a bus-system manufacturer's point of view: Based on the middleware component classes' interface descriptions, platform specific implementations are created for all middleware component classes within a specific connector template (see Section 5.3). Subsequently, a set of contracts (the ones that will later on emerge during middleware synthesis) is created for each implemented middleware component. The component binary, together with its associated contracts is finally stored in a component repository for use in middleware synthesis.

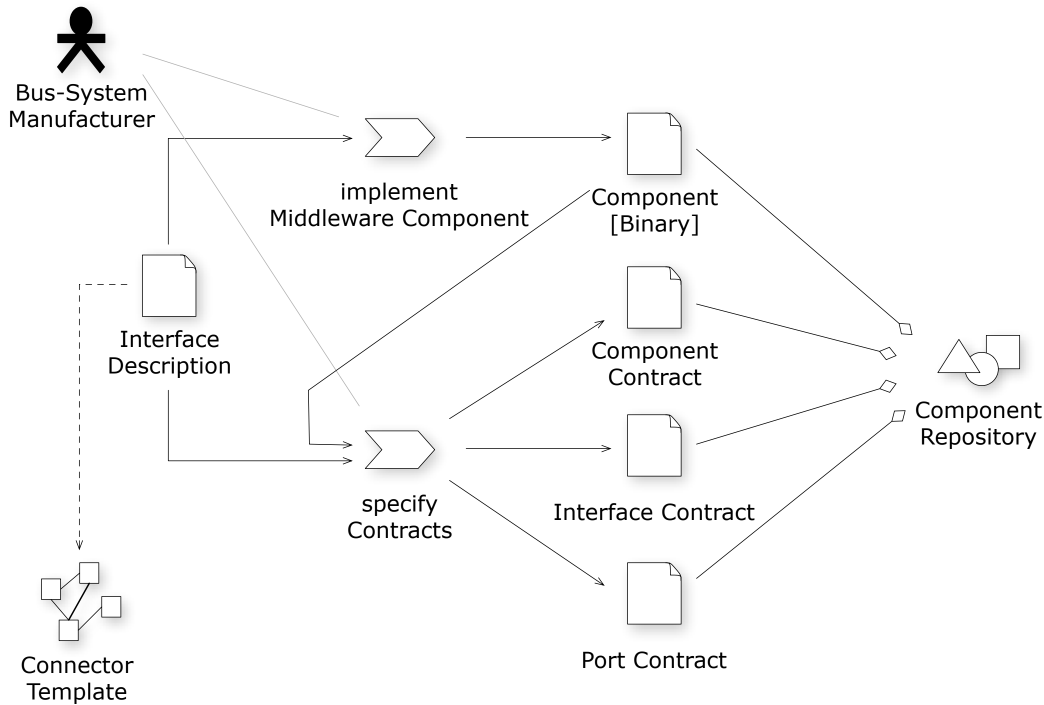


Figure 5.4: Development process for middleware components

In Figure 5.3, six emerging contracts are shown. CF_1 and CF_2 are component contracts, the two contracts labeled P are port contracts, and IA_P and IA_R are interface contracts. Even this simple example exhibits the benefit of emerging contracts: If, for example, the total ROM usage of an application including all required middleware functionality is of interest, it now can be calculated by summing up the ROM usage specified within the four component contracts. In a similar way, end-to-end timing of distributed applications can be calculated (see Section 6.3).

5.3 Connector Transformation

As outlined in Section 2.2, a model driven development process is mainly based on model transformations. This also applies to the *COMPASS* methodology: Custom-tailored communication middleware for *AUTOSAR* applications is synthesized from application models and prefabricated middleware components via the *Connector Transformation* that is a two phase, guided transformation, as depicted in Figure 5.5.

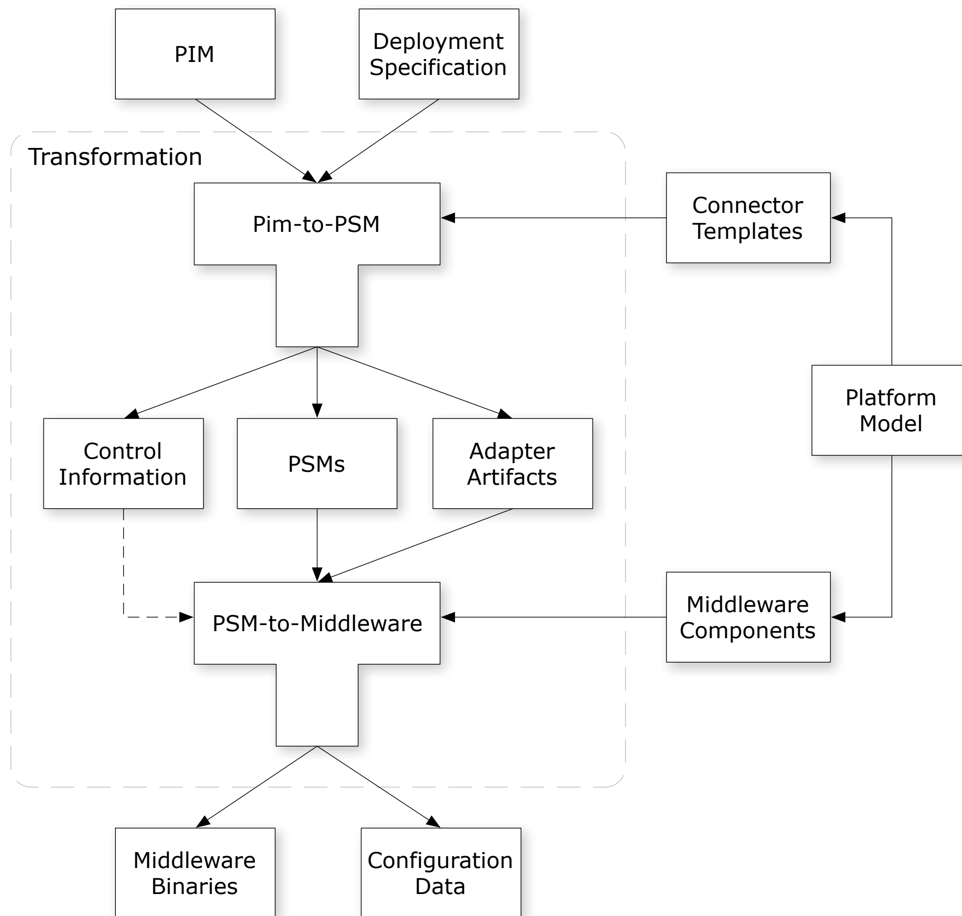


Figure 5.5: Connector Transformation

The first phase, the PIM-to-PSM transformation transforms the platform independent application model into a platform specific model and a set of interface adapter components, for each ECU. In addition it also generates control information (make files, shell scripts, etc.) for the second phase. The transformation process itself is guided by annotated tags—contracts—within the PIM, the application's deployment specification, and architectural middleware patterns, the connector templates. The generated PSMs are component architectures, representing ECU specific views of the distributed application, including node specific middleware component architectures.

The second phase transforms the PSMs into ECU specific middleware binaries for each ECU. In addition, system configuration data is generated in

line with the *AUTOSAR* standard from annotations within the application model and the emerging contracts within the PSMs. The generated binaries finally have to be linked and deployed with the remainder of the software system and the system configuration data in accordance to the *AUTOSAR* methodology, to get an executable automotive application.

5.3.1 Connector Templates

Connector templates are architectural patterns at middleware level for component based communication middleware, and serve as blueprint within the process of middleware synthesis. They are designed as structural templates, thus the placeholders for middleware components are depicted as so called parts. Parts are labeled with the components' role, and are stereotyped as «com primitive»[5]. *Infrastructural service access ports* (see Section 4.2.2) of the depicted parts are optional. Therefore, both, active and passive versions of primitives, can be plugged in at the same place within the structural designs, assuming that the interfaces of associated primitives match.

If a part's functionality is not required for a specific connector[6], a dummy primitive is plugged in instead of a full-fledged one. These dummies satisfy the architecture's interface requirements, but do not provide an implementation. Instead they immediately return, if invoked.

The communication primitives used within the connector templates are coarse-grained architectural abstractions that represent basic functionality within a connector. The provided view resembles parts of the *AUTOSAR Run-Time Environment* and of the *AUTOSAR Basic Software*. However, for the purpose of this thesis the communication subsystem of the *AUTOSAR Basic Software* has been examined and decomposed in a more fine-grained way (see Section 4.3) then expedient for connector templates. Hence, parts within connector templates are meant to be composed artifacts, too. As a result, *COMPASS* middleware components are contained within the *Reader* and the *Writer* communication primitive, described in Section 5.3.1.1. This

[5]Communication primitives represent classes of middleware component assemblies as defined in Section 4.2.2.

[6]Parts like, e.g., buffers may be contained within a template of a connector, but may be omitted, if the contract of the explicit connector within the PIM does not prescribe the parts' presence. In that way multiple optimized variants of one connector type may be described by one single template.

approach in addition decouples design and granularity of middleware components from their final application in communication primitives within connector templates.

5.3.1.1 Sender-Receiver Connector Architecture

The first connector pattern to specify is that of sender-receiver connectors, as client-server connectors can be built upon them. Figure 5.6 shows the architecture of a sender-receiver connector, which can be used to assemble both, state transmission and event transmission connectors in line with *AUTOSAR*.

The depicted connector is made up of two fragments, the sender and the receiver, which have to be deployed independently , aside with the component they are connected to. Depending on the properties of the required connector, full-fledged communication primitives, or light-weight dummies can be plugged into the positions, designated by the template's parts. However, at least one operational writer and one reader has to be plugged into a sound sender-receiver connector. Therefore, both placeholders are outlined fat. Following roles for communication primitives come to use within the defined component architecture:

1. **Reader and Writer**: Communication primitives associated with one of these roles are responsible for the process of information propagation. Writers put information on the information channel, while readers get it from there, whereas the nature of the channel depends on the components' deployment. Both, reader and writer, have to be implemented for any connector class (see Section 5.2.1).

   While readers and writers for local interaction may be implemented as trivial shared memory accesses, they can also become rather complex composed architectures when distributed.

2. **Encoder and Decoder**: Encoders provide functionality that is primarily used by writers to manipulate data before it is emitted to the information channel. Therefore, this role is assigned to communication primitives that provide services of marshaling, data verification, or encryption. Decoders provide services that are required by readers to manipulate data after it has been read from the information channel. The provided services correspond to that of the encoders. Decoder primitives are required only, if encoders are used at the corresponding site of the communication channel.

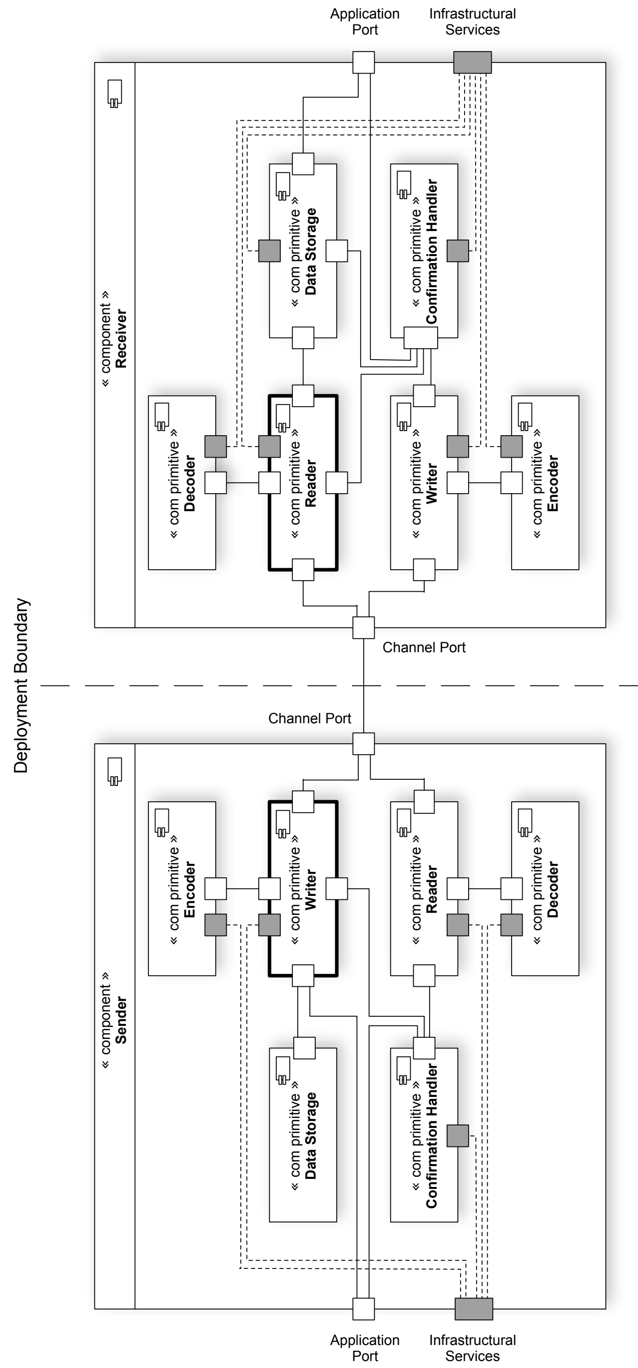


Figure 5.6: Sender-Receiver connector template

3. **Confirmation Handler**: Confirmation handlers are communication primitives that keep control of the application's communication status at *BSW* level. They can provide distinct interfaces for various levels of confirmation. However, only handling of application specific confirmations, like that on successful execution, are within this role's responsibility. Confirmation on successful transmission (TX) and successful reception (RX), is directly handled by the reader and the writer at *Basic Software* level.

4. **Data Storage**: If the receiver fragment has to store received messages, this can be achieved by any communication primitive that implements the data storage role. A message queue is a typical implementation of that communication primitive. In dependable systems, a writer may also store emitted messages to be able to retransmit them in case of a communication error. Parts that implement the data storage role are also used to create and store communication traces, or the node's communication history.

Following the proposed architecture, very simple sender-receiver connectors for data distribution, but also complex ones for event distribution, can be built for each of the three interaction classes. Optimization at architectural level can be achieved by eliminating dispensable parts.

5.3.1.2 Client-Server Connector Architecture

Figure 5.7 shows the schematic for a client-server connector. This connector type utilizes the sender and receiver fragments from Section 5.3.1.1 to realize message propagation along the assigned information channel. Thus, all properties of sender-receiver interaction are also available for the sender and the receiver part within a client-server connector. Additional roles allow the creation of various specific client-server behaviors:

1. **Request Manager**: Within the client fragment, the request manager primitive takes care of the client's service requests. Depending on the connector's needs and the primitive's implementation, various connector behaviors may be implemented. The request manager primitive, e.g., is responsible for the connector's behavior regarding blocking or non-blocking interaction. It may also account response times, and thus may decide if a request has timed-out and hard deadlines are exceeded, which is of great importance in real-time systems.

   The request manager primitive within the server fragment has to invoke the services of the connected server component in accordance to the

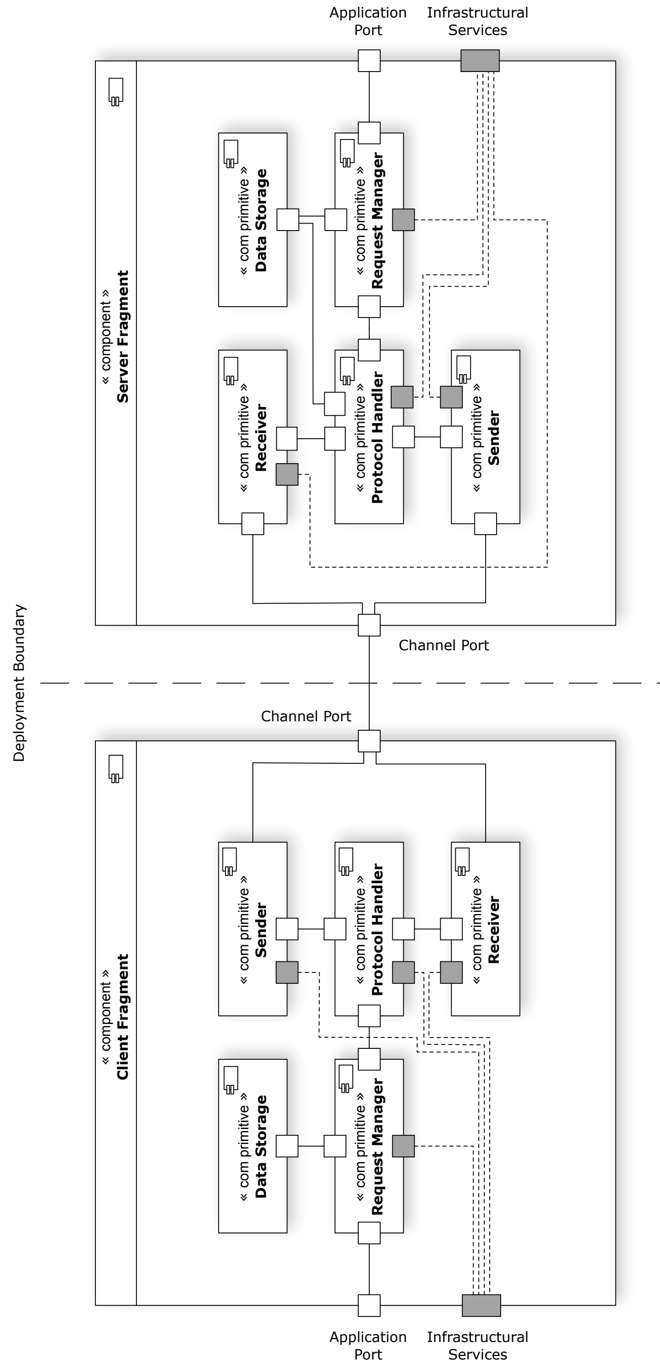


Figure 5.7: Client-Server connector template

messages received from the client fragment. If this primitive has to support concurrent server invocations, it has to be implemented as active primitive, and also has to implement its own job-list for pending invocations.

2. **Protocol Handler**: Protocol handler primitives implement an arbitrary communication protocol at *Run-Time Environment* level for client-server interaction.

   Protocol handler primitives, as well as request manager primitives, only operate in corresponding pairs. Hence, they have to be enclosed in neither or in both fragments of a connector, accordingly.

3. **Data Storage**: At the server-side connector fragment, data storage primitives can be used to queue incoming but not-yet served requests. At the client-side fragment, sent requests are stored for monitoring and accounting of asynchronous calls. The primitives on both sides are optional, and are not related to each other. The data storage element within client-server connectors differs from that within sender receiver connectors mainly by its purpose, not by its type. A client-server connector for example, which has to store time-stamps of service requests, but does not buffer incoming messages, would not contain a full-fledged data storage within its receiver, but would enclose one itself.

4. **Sender and Receiver**: Both communication primitives are composed middleware components. They are built from *AUTOSAR Basic Software* middleware components, like the ones discussed in Section 4.3, and thus primarily encapsulate *Basic Software* functionality.

   As client-sever interaction is based on sender-receiver communication, each client-server connector contains at least one sender-receiver connector.

Using this architectural template, a wide number of client-server connectors, like simple procedure-calls (local and remote), or even real-time capable method invocations can be constructed.

5.3.2 Connector Transformation Algorithm

Based on the *COMPASS* component model and the connector templates provided in Section 5.3.1, Algorithm 2 provides the *Connector Transformation*, which synthesizes application specific communication middleware: In a first

step, one PSM is created for each ECU of the distributed system; application components are stored within those ECU specific PSMs in accordance to the application's deployment specification. In a second step, each connector within the platform independent application model is examined with respect to the deployment scenario. As a result, deployment and platform specific connector fragments are selected in accordance to interaction specific requirements, and are also stored and locally connected within the PSMs.

The used formalism in Algorithm 2 denotes sets starting with a capital letter and ending with curly brackets as subscript. Elements are denoted starting with a lowercase. To make the code more readable, tuples as much as specific positions within tuples are identified by names. The following definitions provide structure and names for the tuples and sets used within the algorithm's description.

The *Connector Transformation* requires a platform independent application model and a deployment specification as input, and creates a platform- and node-specific component architecture for each system node that is part of the application.

Definition 5 (PIM). *The application's component architecture is described by a platform independent model, denoted by a pair of sets of application components and of explicit connectors that connect the components.*

$$\text{pim} = (\mathit{Components}_{\{\}}, \mathit{ExplicitConnectors}_{\{\}}) \tag{5.1}$$

Definition 6 (Connector). *A connector is a tuple containing a client component, a server component, the interface used for the connection, the connector's interaction paradigm, and a contract for the connector, for example WCET annotations for the interface. Both, explicit connector and local connector, are isomorphic to the tuple* connector.

$$\text{connector} = (\text{client}, \text{server}, \text{interface}, \text{iparadigm}, \text{contract}) \tag{5.2}$$

$$\text{explicitConnector} \cong \text{localConnector} \cong \text{connector} \tag{5.3}$$

Definition 7 (Deployment Model ($\mathbf{DM}_{\{\}}$)). *The application's deployment model describes where the application components are deployed to. Valid*

locations for deployment are described by a location *that is a pair of ECU and address space. The application's deployment model is represented by a set of pairs of one location and one set of components deployed to that location, such that there exists exactly one pair for each location of the target system.*

$$DM_{\{\}} = set\ of\ (\text{location}, Components_{\{\}}) \tag{5.4}$$
$$\text{location} = (\text{ecu}, \text{addressspace}) \tag{5.5}$$

Definition 8 (**ConnectorPatterns$_{\{\}}$**). *The set of connector patterns contains one architectural pattern for each interaction style (* distributionType *and* iparadigm*).*[7] *A connector pattern is a pair of connector fragments, one for the client- and the server-side each. A connector fragment is a tuple containing a set of component classes (place-holders for concrete middleware components), a set of local connectors, connecting the middleware components, and an interface that denotes the fragment's generic interface type.*

$$ConnectorPatterns_{\{\}} = set\ of\ (\text{distributionType}, \text{iparadigm}, \text{connectorPattern}) \tag{5.6}$$
$$\text{connectorPattern} = (\text{clientFragment}, \text{serverFragment}) \tag{5.7}$$
$$\text{connectorFragment} = (ComponentClasses_{\{\}}, LocalConnectors_{\{\}}, \text{interface}) \tag{5.8}$$
$$\text{clientFragment} \cong \text{serverFragment} \cong \text{connectorFragment} \tag{5.9}$$

The connector patterns do not prescribe specific middleware components but classes of compatible components. The selection of a concrete implementation of a prescribed component class allows to flexibly adjust the resulting middleware to specific application needs.

Definition 9 (**MWComponents$_{\{\}}$**). *The set of available middleware components provides the concrete building blocks, which may be assembled to form the synthesized middleware. The set contains tuples of middleware components and associated contracts.*

$$MWComponents_{\{\}} = set\ of\ (\text{component}, \text{contract}) \tag{5.10}$$

[7] To support multiple vendors and variants, the set of connector patterns has to be a multi-set, capable of containing multiple patterns of same type.

The algorithm's output is specified as:

Definition 10 ($\mathbf{PSM}_{\{\}}$). *The algorithm returns a set of platform specific models, one for each system node. These models are affiliated with exactly one location, and contain all components (application and middleware) that are deployed to that location. Additionally, the models contain a set of local connectors, which connect the components.*

$$PSM_{\{\}} = set\ of\ \text{psm} \tag{5.11}$$
$$\text{psm} = (\text{location}, Components_{\{\}}, LocalConnectors_{\{\}}) \tag{5.12}$$

Taking the given definitions into account, the algorithm of the *Connector Transformation* can be described as follows[8]:

The *Connector Transformation* starts with creating and initializing one PSM for each system node (Algorithm 2, line 3-4). Then, the main loop iterates over all explicit connectors, which is a quintuple contained within the set $ExplicitConnectors_{\{\}}$ of the PIM (Algorithm 2, line 5). Note the way of binding variables (e.g. *client*) to specific positions within the tuple. The deployment situation of all connected components is determined (Algorithm 2, line 6-9) to derive proper middleware requirements.

If the component pair is co-located, a local connector is added to the PSM that contains the two components (Algorithm 2, line 14), elsewise a connector pattern is selected with respect to the connector's interaction paradigm and the application components' deployment scenario (Algorithm 2, line 21). As connectors for distributed interaction consist of two fragments, both fragments, together with their associated application component and with location information, are stored within a work-set (Algorithm 2, line 22-24).

To materialize the explicit connector for distributed interaction, the following procedure is applied to both elements within the work-set: For each part of the pattern of the connector fragment a concrete implementation of a feasible middleware component is selected from the set of available middleware components (Algorithm 2, line 28). Feasibility is determined by matching the fragment's contracts with that of the middleware component. A fragment is

[8] As the *Connector Transformation* mainly takes place at model level, the term component is used for $M1$ model level elements, and must not be mixed up with $M0$ component implementations. The same applies to connectors, communication primitives, etc.

Algorithm 2: Connector Transformation

```
1  ConnectorTransformation(pim, DM{}, ConnectorPatterns{}, MWComponents{} ):PSM{}
2  begin
       // create and initialize PSM for each location
3      PSM{} ← ∅;
4      foreach (location, Components{}) ∈ DM{} do  add (location, Components{}, ∅) to PSM{};
       // iterate over all explicit connectors within the PIM
5      foreach (client, server, interface, iparadigm, contract) ∈ ExplicitConnectors{} of pim do
6          foreach (location, Components{}, LocalConnectors{}) ∈ PSM{} do
7              if client ∈ Components{} then  clientLocation ← location;
8              if server ∈ Components{} then  serverLocation ← location;
9          end
10         MiddlewareComponents{} ← ∅; CFCons{} ← ∅;
11         if clientLocation == serverLocation then    // components are co-located
12             distributionType ← LOCAL;
13             select psm ∈ PSM{} | location of psm == clientLocation;
14             add(client, server, interface, iparadigm, contract) to LocalConnectors{} of psm
15         else
16             if ecu of clientLocation == ecu of serverLocation then
17                 distributionType ← INTERPROCESS;
18             else
19                 distributionType ← REMOTE;
20             end
               // select apropriate connector pattern
21             select (d, i, selectedPattern) ∈ ConnectorPatterns{} | d ==
               distributionType ∧ i == iparadigm;
               // compute fragments for client and server location
22             WorkSet{} ← ∅;
23             add (client, clientLocation, clientFragment of selectedPattern) to WorkSet{};
24             add (server, serverLocation, serverFragment of selectedPattern) to WorkSet{};
25             foreach (appComp, cfLocation, connectorFragment) ∈ WorkSet{} do
26                 (ComponentClasses{}, CFCons{}, cfInterface) ← connectorFragment;
                   // select optimal implementation for each component class
27                 foreach part ∈ ComponentClasses{} do
28                     select (mwc, contract′) ∈
                       MWComponents{} | mwc implements part∧contract′ satifies contract;
29                     add mwc to MiddlewareComponents{};
30                     if mwc comprises cfInterface then
                       fragmentRoot ← mwc;    // for interface adapter
31                 end
                   // add interface adapter
32                 iAdapter ←GenerateInterfaceAdapter(cfInterface, interface);
33                 add iAdapter to MiddlewareComponents{};
34                 add (appComp, iAdapter, interface, iparadigm, contract) to CFCons{};
35                 add (iAdapter, fragmentRoot, cfInterface, iparadigm, contract) to CFCons{};
                   // add middleware components and local connectors to PSMs
36                 select psm ∈ PSM{} | location of psm == cfLocation;
37                 add MiddlewareComponents{} to Components{} of psm;
38                 add CFCons{} to LocalConnectors{} of psm;
39             end
40         end
41     end
42 end
```

a composed architecture that contains exactly one part, which provides the fragment's external interface—the one to which the application component is bound to. This dedicated part is stored as the fragment's root (Algorithm 2, line 30). If no proper implementation is available, the transformation can not be finished successfully[9]; if more than one candidate for selection exists, the one with best overall properties can be selected[10].

Subsequently, interface adapter components are generated, which connect the application component to the generic connector interface exposed by the fragment's root (Algorithm 2, line 32-35). Finally, the middleware components and local connectors are added to the PSM that contains the fragment's associated application component (Algorithm 2, line 36-38).

5.3.3 Model-to-Code Transformation

As shown in Figure 5.5 on page 78, an additional model-to-code transformation (PSM-to-Middleware) is required to gain executable binaries for each ECU. However, the model-to-code transformation for component based architectures is straight-forward, and thus is outlined here on a basic level only.

The provided algorithm of the *Connector Transformation* calculates platform specific models for each system node. In addition, interface adapter components are generated, which glue the application components to the generic interfaces of utilized middleware components. In consequence, after successfully executing the *Connector Transformation*, a platform specific model, and binaries of all components within the PSMs exists for each ECU.

Based on the PSMs, the model-to-code transformation generates control files, which bind all components that are deployed to the same ECU. This binding is done by modifying the binary components' function tables and consecutively linking the modified binaries.

The modification of the function tables results from the fact that calls to externally provided functions refer to abstract function names. These abstract names contain the interface type that exposes the called function. A call within component A to a required external function f, which is contained within an interface I, is denoted as call to *I::f* within A's source code.

[9]To solve this issue, either additional middleware component implementations have to be made available, or communication requirements have to be changed.

[10]The wording *best overall property* implies the existence of some kind of metric for component performance.

However, f may be provided by any arbitrary component that implements the interface *I*, and that is not known at compile-time of *A*. The concrete provider of f becomes known after specifying the architecture's composition, e.g. component *B*. Consequently, the abstract call to *I::f* is rerouted to *B*'s implementation *B::f*, by patching *A*'s function table.

In addition, the model-to-code transformation extracts system configuration data from contracts within the PIM and the PSMs. This data is finally deployed to the system's ECUs in conjunction with the executable.

5.4 Summary

This chapter described a model driven methodology for automatic synthesis of component based communication middleware within the domain of *AUTOSAR*.

So called explicit connectors that store interaction specific properties in application models were established as first class architectural entities. In addition, the concept of emerging contracts was introduced to enable model level validation of synthesized middleware architectures. Emerging contracts contain middleware specific properties that originate from assembled middleware components, and are provided by the middleware components' manufacturers.

As model transformations are the core of model driven development, this chapter's main contribution is the so called *Connector Transformation* that transforms PIMs into PSMs. It translates explicit connectors within platform and deployment independent application models into communication specific component architectures within the application's platform specific models. These middleware component architectures are based on the *COMPASS* component model for *AUTOSAR Basic Software*, described in Chapter 4. To gain executable binaries for each ECU of the distributed application, the generated PSMs are finally transformed into code. The generated communication subsystem is fully compatible to the *AUTOSAR* standard in terms of its external interfaces. However, it is custom-tailored and hence light-weight compared to a conventional *AUTOSAR* communication subsystem.

To demonstrate the benefit of the described approach, a typical automotive application was implemented twice: the first version is developed in accordance to the traditional *AUTOSAR* methodology, the second version in line with the methodology described within this thesis. Detailed measurements

are provided in Chapter 6 and inter alia show a reduction of the communication subsystem's memory footprint of more then 30%.

Chapter 6

Discussion

The following sections provide an evaluation of the *COMPASS* methodology and of the algorithms proposed within this thesis. In addition, related work, as far as not covered within the previous chapters, and future work is discussed.

6.1 Proof of Concept

To prove the concept of component based *AUTOSAR* middleware, and to evaluate the gained benefit, three consecutive steps are taken: First, the static *Cohesion Analysis* (see Section 4.3.2) is applied to a full-fledged industrial implementation of a conventional *AUTOSAR* communication stack, in order to gain *COMPASS* middleware components. Second, a prototypical automotive application is created in line with the *COMPASS* methodology to provide a testbed for automatic middleware synthesis. Third, custom-tailored communication middleware for the application from step two is synthesized via the *Connector Transformation* (see Section 5.3) using the middleware components from step one.

The resulting custom-tailored *Basic Software* is finally compared to its conventional industrial counterpart to clearly show the advantages of automatically synthesized COMPASS communication middleware in terms of ROM footprint and CPU usage.

6.1.1 Middleware Component Recognition

To obtain *AUTOSAR* compliant middleware components, a layered reference implementation of the *AUTOSAR BSW* communication subsystem is

FlexRay Interface			
	# of files	LOC	kB
Header	4	1620	59
Implementation	15	4192	135
FlexRay Driver			
	# of files	LOC	kB
Header	13	1660	88
Implementation	27	7142	222

Table 6.1: Base characteristics of the analyzed communication stack

decomposed via the static *Cohesion Analysis*.

The analyzed source code is full-fledged C code that implements the *FlexRay Interface- and Driver-Layer* (see Section 3.3.2). The generated abstract syntax tree (AST) contains 291794 nodes, and was traversed once by the algorithm. The full execution (including I/O) of the *Cohesion Analysis* algorithm took less than 3 seconds, and used approx. 80MB of memory on a dual 64-bit Core 2 Duo Xeon workstation at 3.0 GHz.

The source code of the implementation can be characterized as shown in Tabel 6.1 and in Tabel 6.2:

- **Base Characteristics:** The implementation of the *FlexRay Interface* module is realized in 4 header (.h) and 15 implementation (.c) files with a total of 5812 lines of code. The *FlexRay Driver* module is realized in 13 header (.h) and 27 implementation (.c) files with a total of 8802 lines of code. Accordingly, the analyzed code base consists of 59 files containing 14614 lines of code (504 kB).

- **Code Characteristics:** The characteristics of the analyzed implementation, the *FlexRay Interface* module and the *FlexRay Driver* module, is summarized in Tabel 6.2. As denoted, the implementation consists of 107 functions that in total contain 431 calls to other functions. This number refers to the number of call-sites and not the number of calls executed at run time, as call expressions may be located within loops or within conditional branches of the program. The rather high number of 4924 cast expressions results from the fact that all—explicit as much as implicit—cast operators are contained within the AST generated by the implemented analyzer.

Characteristic	# of occurrences
Function Definitions	107
Function Call Expressions	431
AddressOf Operators	12
Ptr Deref. Expressions	241
Arrow and dot Operators	457
Cast Expressions	4924

Table 6.2: Code characteristics of the analyzed communication stack

As required by the *Cohesion Analysis*, a set of 87 functions has been marked as relevant.[1] 50 data fields within 12 data structures have been marked as irrelevant, and thus have been filtered out by the analysis. A detailed list of all functions marked as relevant, and all structure fields marked as irrelevant is provided in Appendix D.

Figure 6.1 provides a visualization of the analysis result of the *Cohesion Analysis* algorithm. The diagrams contain structure fields denoted as boxes, and functions denoted as circles. Figure 6.1a shows a full unprocessed component graph of the *Basic Software* under analysis. Figure 6.1b shows the final result of the *Cohesion Analysis* applying the filters for marked functions and data fields. The clusters denoted in Figure 6.1b represent those component classes that have been identified by domain experts via manual decomposition. Strong cohesion only exist within component classes (the clusters), while no edges exist between distinct component classes.

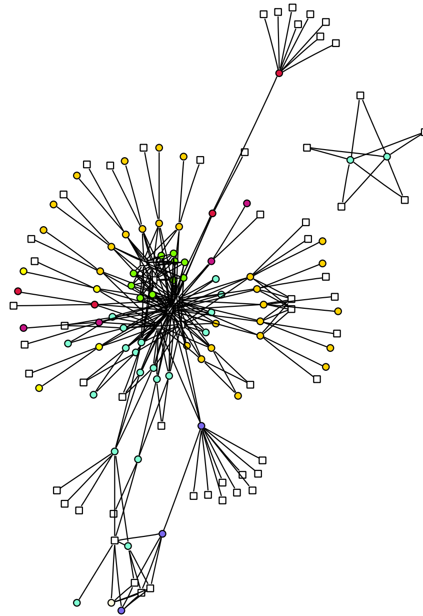
6.1.1.1 Comparison of Approaches

Manual decomposition identified eight component classes, as described in Section 4.3.1: *Base*, *Transmitter*, *Receiver*, *WUP*, *Time Service*, *Status*, *Media Test*, and *Transceiver Driver*.

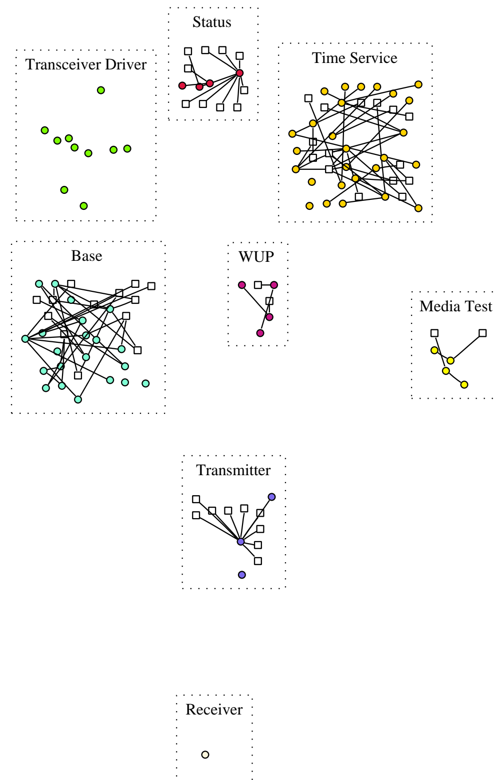
Compared to the manually gained decomposition, the *Cohesion Analysis* is able to recognize not only one, but a set of valid decompositions. These multiple valid results are caused by the fact that too small components may by grouped into one bigger and hence more practicable component.

By looking at the *Transceiver Driver* component class or at the *Receiver* component class, it becomes clear why the algorithm provides not just one

[1] Unlike proposed in Chapter 4 relevant functions have been marked, as *COMPASS* middleware has to adhere to the *AUTOSAR* standard and its prescribed interfaces.



(a) Unprocessed component graph



(b) Analysis result

Figure 6.1: Recognized middleware components for *AUTOSAR*

but a whole set of solutions: Each function contained within the manually identified class *Transceiver Driver* is completely uncoupled. Hence, the proposed clustering is thetic. Each of those functions could be moved to an arbitrary cluster without violating the algorithm's constraints for component classes. In fact, any permutation of uncoupled components into groups leads to a valid result in terms of cohesion.

Both, manual decomposition and the *Cohesion Analysis*, depend on configuration specific data like described in Section 4.3.2.2. As a consequence, repetitive analyses for modified configurations have to be done.[2] However, the process of marking functions and structure fields by domain experts for the *Cohesion Analysis* has to be done only once. As the *Cohesion Analysis* is performed semi-automatically, its ability to scale to multi-configuration analysis clearly outnumbers the manual approach in terms of required expert resources.

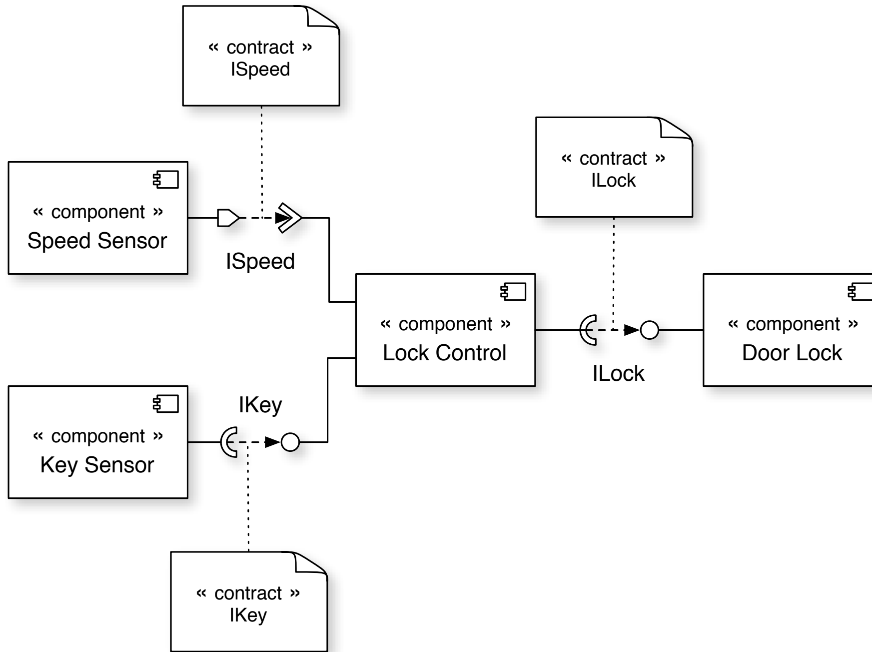
6.1.2 Testbed Application

To prove the model driven methodology proposed within this thesis, it is applied to the development process of an *AUTOSAR* compliant application. The application chosen as testbed is a typical distributed automotive application: a cruise-aware door-lock application. The main purpose of the application is to lock and unlock a vehicle's doors. This should be done on user intervention by utilizing a key, or automatically in accordance to predefined rules and the vehicle's current speed. All doors should automatically be locked, if the vehicle's speed exceeds a given limit. Once the door is locked, no unlock operation must be performed while the vehicle is moving. The application is simplified for demonstration purpose, but nevertheless is focused on the essential issues of a distributed automotive application. To cover the topics of distributed and local interaction, the application incorporates event-based program logic as much as time-driven communication facilities.

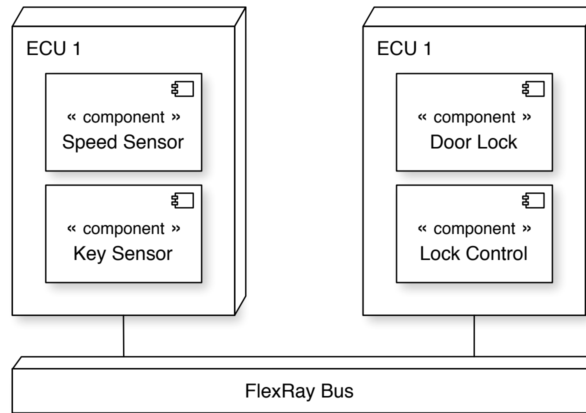
Figure 6.2a describes the application's platform independent component architecture, containing all components and all explicit connectors. The door lock application is made up of four application components:

1. **Speed Sensor**: The speed sensor component provides the vehicle's current velocity by its interface *ISpeed*. The interface is a sender-receiver interface, which is denoted in line with the *AUTOSAR* UML

[2] Minor changes within preprocessor directives (e.g., conditional compilation) may lead to major changes in the overall system's structure.



(a) Component architecture



(b) Deployment specification

Figure 6.2: Automotive door-lock application

profile. The component will broadcast the vehicles current speed periodically, which is a timing constraint of the component but which is also denoted in the interface contract *ISpeed*.

2. **Key Sensor**: The key sensor component handles any key activity of the user. It reports a lock or unlock action to the lock control component by invoking the *lock()* or *unlock()* method of the *IKey* interface.

3. **Lock Control**: The main application logic resides within the lock control component. This component receives the vehicles current speed by its sender-receiver interface *ISpeed*, and all key related user activities by its client-server interface *IKey*. The component controls the vehicle's door lock, by invoking the *lock()* or *unlock()* method of the *ILock* interface.

4. **Door Lock**: This component locks or unlocks the vehicle's doors. It implements the functions *lock()* and *unlock()*, and provides them within the *ILock* interface.

The contracts for the explicit connectors are depicted as *ISpeed*, *IKey*, and *ILock*. Within these contracts, communication and interaction specific attributes are specified; the *ISpeed* contract for example defines that no confirmation for successful reception is required.

The application's deployment scenario is provided in Figure 6.2b. To test local as much as distributed interaction, the components are deployed onto two ECUs, ECU_1 and ECU_2 (both of type ARM 922T, providing 16 bit access to an ERAY 1.0 *FlexRay* communication controller).

6.1.3 Middleware Synthesis

After successful refactorization of an *AUTOSAR* communication stack as described above, application- and node-specific communication middleware for the proof-of-concept application is synthesized via the *Connector Transformation*.

Following the algorithm provided in Chapter 5, the application's PIM is transformed into two PSMs—one for each ECU—with respect to the specified deployment scenario. To inject proper connector fragments into these PSMs, all three explicit connectors within the PIM are classified in accordance to their interaction style, communication requirements, and their distribution scenario:

1. **Connector at *ISpeed* Interface**: This connector is an XN sender-receiver connector. Its *RTE* as much as its *BSW* functionality is constructed in accordance to the sender-receiver template provided in Section 5.3.1. As specified within the connector contract *ISpeed*, no confirmation for reception is required. Thus, only reader functionality at the receiver node, and writer functionality at the sender node is required for this simple "data broadcast" connector.

2. **Connector at *IKey* Interface**: The connector for the *IKey* interface is an XN client-server connector. It is assembled in accordance to the client-server template. The protocol handler, which in terms of *AUTOSAR* is part of the RTE, implements a simplified version of the RPC protocol [BN84, BALL89, ATK92], thus requires sender- and receiver-functionality on the client as much as on the server side.

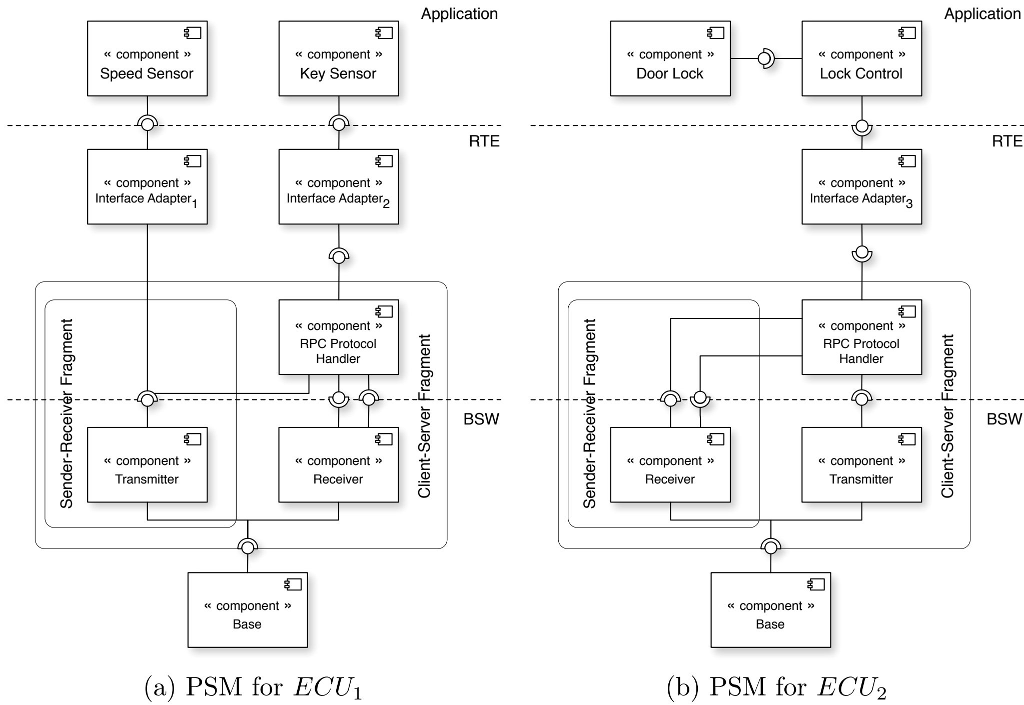
3. **Connector at *ILock* Interface**: Due to the co-located deployment of the *Lock Control* and the *Door Lock* component, this connector is a local client-server connector. By applying the proposed transformation algorithm, this connector is transformed into a local connection, a simple function call.

Figure 6.3 depicts the two PSMs describing the application's static component architecture after successful transformation. Both models contain the full set of components that have to be deployed to each specific ECU:

1. **Application components**: *Speed Sensor* and *Key Sensor* for ECU_1, *Door Lock* and *Lock Control* for ECU_2.

2. **RTE components**: As described in Chapter 3, the *RTE* glues the application components to the generic *BSW*. In addition, the *RTE* implements interaction behaviors and protocols like signal distribution and remote procedure calls. As a result, the PSMs contain middleware components that provide *RTE* functionality: $InterfaceAdapter_1$ and $InterfaceAdapter_2$ for ECU_1, and $InterfaceAdapter_3$ for ECU_2 are generated from the application components' interface specifications within the PIM, and from the *AUTOSAR* compliant specifications of the *RTE* interfaces. The *RPC Protocol Handler* component on both ECUs implements remote procedure calls in line with the *AUTOSAR* specification.

3. **BSW components:** Due to the minimalist application requirements on the *Basic Software*, only three out of eight *BSW* components are



(a) PSM for ECU_1

(b) PSM for ECU_2

Figure 6.3: Result of applied model transformation.

required on each ECU. Only the *Transmitter*, the *Receiver*, and the mandatory *Base* component have to be deployed for a fully functional application. The remaining five *BSW* components are not required by the application, and thus are simply omitted.

To obtain executable binaries for each ECU, a simple PSM-to-code transformation is performed: Based on the PSMs, source code for the adapters is automatically generated and compiled into adapter components. Thereafter, the function-tables of all component binaries are patched as described in Section 5.3.3, in order to map abstract interface-related calls to concrete implementations of providing components. Finally, all components and all additional system binaries (e.g., OS) are linked, using generated make-files.

To demonstrate the optimization capabilities of the *COMPASS* approach, several variants for some of the identified middleware component classes have been built. These variants differ in size and functionality, and hence allow a demand-specific reduction in terms of size of an application-specific middleware component architecture. By examining the connector contracts within the PIM, the *Connector Transformation* is able to select the slimmest variant of each component class that fulfills the contract. The variants built and

Component Class	Variant	ROM Usage
Base	standard	7899
	no cluster functionality	5803
Media Test		1108
Receiver		1188
Status		1364
Time Service	standard	4280
	absolute	1616
	relative	1612
	global	1052
Transceiver Driver		4528
Transmitter		1392
WUP		860

Table 6.3: BSW components for FlexRay

used for the purpose of this thesis are specified in detail in Appendix C, and are summarized in Tabel 6.3.

When comparing the created software system—application and ECU specific middleware—to its conventional layered counterpart, the synthesized component based middleware proves to be more efficient: In best case, the conventional communication stack containing all functionality requires a total of 21.512 bytes, whereas a total of 6.884 bytes has been eliminated within the component based version. Thus, the *COMPASS* approach gains a reduction in size by more than 30%. In addition, a run-time performance increase (10% less CPU usage) is gained. Detailed comparison of execution paths of both middleware versions showed that the strict layering of the conventional *BSW* enforces numerous cascaded delegation calls to access functionality of lower layers. By breaking up this layering via the component based design, these functions become directly accessible, which eliminates most delegation calls.

Finally, an alternate application scenario has been specified to examine the methodology's limitations, where all types of interaction occur on all ECUs. By requiring the full set of communication functionality within each node, optimization is prevented intentionally. However, in this worst case the synthesized middleware shows an equivalent memory footprint and run-time performance than the conventional one.

6.2 Related Work

The following section summarizes related work that is of relevance for this thesis but has not explicitly been discussed so far. Section 6.2.1 provides a brief discussion of well established component models, Section 6.2.2 describes work on static analysis with respect to relevance for the proposed component recognition algorithm, and Section 6.2.3 finally describes work on model driven development and model transformations. The domain of *AUTOSAR* is not covered here, as it is extensively described in Chapter 3.

6.2.1 Component Models

The *KOALA* component model [vOvdLKM00, vO98, vO04] was developed by Van Ommering, Van der Linden, Kramer, and Magee, for embedded systems within the domain of consumer electronics products. *KOALA* is a hierarchical model—composed components are called compound components—that aims at software reuse and support for product-lines. Components are encapsulated pieces of software and architecture, which are described in a text based component description language (CDL). Using the CDL *KOALA* handles diversity on internal as much as on external level. Internal diversity aims at configuration specific variants of components, whereas external diversity aims at optimization at architectural level. *KOALA* components are implemented in C, whereas function names contain a sub-string that encodes the component and the port to which the function belongs to. Component architectures are specified using a proprietary architecture description language (ADL) to manage product complexity. To connect provided and required functions of two components in accordance to the architectural description, preprocessor based string replacement is used at source code level to substitute calls to abstract functions[3] within the requiring component, by calls to concrete implementations within the providing component.[4] To optimize component architectures, as much as to rebind them, the components' source code has to be available in order to rerun the preprocessor and to recompile the components. Although the *KOALA* and the *COMPASS* component model have much in common, they differ in important aspects: *COMPASS* aims at the domain of *AUTOSAR*. As a result it relies on UML 2.0 profiles to describe component interfaces as much as component architectures. To statically bind

[3] A concrete implementation of the called function is typically not known at a component's implementation time.

[4] The usage of the C preprocessor for configuration specific code-rewriting is common practice within the embedded systems domain.

two connected components, *COMPASS* does not necessitate the components' source code. Static binding is achieved by patching function tables within the components' binaries. In consequence, third party component developers do not have to disclose their intellectual property (the source code) when selling their components.

The *SOFA* [SOF09] component model was developed by Plasil et al. [BP01, Bál02, BP04] for distributed desktop and enterprise applications. It is a hierarchical component model, hence components may be nested or be primitives. *SOFA* components are described by their frame and their architecture. The frame specifies the components' external interfaces and configuration parameters; the architecture specifies the internal structure of a nested component. One specific frame may be implemented by various architectures. Components are described in a component description language that is based on OMG's Interface Definition Language (IDL). In *SOFA*, connectors are first class architectural entities, which are still present within a running application (in contrast to explicit connectors in *COMPASS*). *SOFA* defines three types of basic connectors: *Procedure Call*, *Event Delivery* and *Data Stream*. In addition, user-defined connectors can be specified. Communication between *SOFA* components can be captured formally via so called behavior protocols. A behavior protocol is a regular-like expression on the set of all (function call) events, which are exchanged between connected components. *SOFA* distinguishes between interface protocols and frame protocols. The concept of explicit connectors in *COMPASS* is based on the idea of *SOFA* connectors. However, as *COMPASS* is targeted at resource constrained embedded system, explicit connectors have no specific implementation at run-time. They are first class architectural elements at model level, but are transformed into local invocations of the system's communication middleware.

The *FRACTAL* [BCS04] component model was developed by Bruneton, Coupaye, and Stefani. *FRACTAL* targets at rather opposite domains, like embedded systems and enterprise applications for telecommunication, by defining an extensible component model. The definition of the term component ranges from a very basic level, at which a component is defined to be e runtime entity that does not provide any control capability to other components, up to more complex levels, where components have to provide means of introspection, configuration, and modification. A *FRACTAL* component is specified either within an interface definition language, or directly at source code level. The *FRACTAL* component model also contains connectors, which are however of secondary nature only. A so called binding is defined to be a com-

munication path between component interfaces. Bindings are classified to be primitive or composite. A primitive binding connects one client interface and one server interface within the same address space. A composite binding is a communication path between an arbitrary number of distributed component interfaces, and is represented as a set of primitive bindings and binding components. Binding components are called *FRACTAL* connectors, and are regular *FRACTAL* components, whose role is dedicated to communication. As connectors are of no primary concern in *FRACTAL*, no further specification on interaction is provided. Explicit connectors in *COMPASS* inter alia represent communication paths, but in contrast to *FRACTAL* bindings allow detailed specifications of interaction specific properties for middleware synthesis and middleware configuration.

6.2.2 Static Analysis

Lee et al. [LSK+01] describe a methodology to recognize components within existing object-oriented source code considering class cohesion. They propose to calculate coupling by message passing and data usage, and in addition consider coupling by class association, composition and inheritance. To keep their analysis effort small, their approach relies on domain knowledge, mainly extracted from UML use-cases and architectural descriptions. In contrary, the *Cohesion Analysis* algorithm described within this thesis is intended to work on object-oriented but also on non-object-oriented source-codes[5]. It also relies on domain knowledge, which is represented by marks on functions and data structures, to keep the analysis's effort as low as possible.

Emami, Ghiya, and Hendren [EGH94] present a context-sensitive approach that generates a graph representing all invocation paths (in the absence of recursion), and precisely handles indirect calls through function pointers in C. They focus on analysis of stack-directed pointers, and collect alias information in the form of points-to relationships. They claim through empirical evidence that exponential behavior is not seen in practice and suggest the use of a memorization scheme to avoid redundant analysis. The *Cohesion Analysis* algorithm described within this thesis is linear in size of the program, but requires function pointers to be pre-processed (annotated) in case of ambiguity. The integration of points-to information is not covered within this thesis, but is subject to ongoing research (see Section 6.3). Nevertheless, the proposed algorithm's precision is sufficient to calculate the required properties of the input program.

[5]The source code analyzed within this thesis is a full-fledged C code.

The concept of abstract memory locations described in Chapter 4 was successfully applied within the following two publications: Hind, Burki, Carini, and Choi give experimental results of comparing flow-sensitive and flow-insensitive flow analysis algorithms in [HBCC99] based on memory locations associated with names. The call graph is constructed while alias analysis is performed in a similar way to the *Cohesion Analysis* algorithm. Diwan, McKinley, and Moss evaluate three alias analysis algorithms based on programming language types. The most precise of these three is a flow-insensitive analysis that uses type compatibility and additional high-level information such as field names [DMM98]. They use redundant load elimination to demonstrate the effectiveness of the algorithms in terms of opportunities for optimization.

6.2.3 Model Driven Development

In [RSB+04] Rackel et al. describe a model driven component based methodology as used by the BMW group. Although the described process is targeted at the domain of enterprise and business applications, it is comparable to the *COMPASS* approach, described within this thesis. Platform independent models are used to define relations between application components in terms of interfaces and connections. Using model transformations, platform specific models are generated from the PIMs. Platform specific architectural issues are not visible within PIMs, and are injected into the PSMs by model transformations. The PSMs finally are transformed into source code for further use. In *COMPASS* an application is also specified via a platform independent model, and in addition by a deployment model. The *Connector Transformation* automatically generates PSMs from the PIM, the deployment specification, and a set of available platform specific middleware components. A last step finally creates an executable binary from middleware components, compiled interface adapters, and application components. Both approaches require an application to be modeled at PIM level, while PSMs and executables are generated by model transformations.

Gokhale et al. outline the integration of MDA into the development process for distributed embedded real-time systems in [GSL+03]. The paper lists various alternatives on how to benefit on MDA. Full middleware synthesis as proposed within this thesis is described, however it is not chosen by the authors for their project (CoSMIC). As they aim at a QoS-enabled CORBA Component Model implementation, they focus on the synthesis of configurations and profiles, as much as on the synthesis of QoS-enabled application

functionality in component assemblies. In contrast, *COMPASS* aims at the provision of component based *AUTOSAR* middleware, and hence is focused on full middleware synthesis.

In [ASSK02] Agrawal et al. describe a technique to transform platform independent models into platform specific models using graph transformation. Both, PIMs and PSMs are considered to be graphs. Hence, PSMs are generated by transforming the PIMs' graphs: Input patterns, sub-graphs within the PIM, are transformed into refined output patterns, sub-graphs within the PSM, via predefined rules. The *Connector Transformation* specified within this thesis works in a similar way: Input patterns within the PIM, the explicit connectors, are transformed into platform specific component architectures within the PSMs (see Chapter 5).

6.3 Future Work

Based on the concepts presented within this thesis, future and ongoing research aims at three directions for the automotive domain:

- By extending the *Cohesion Analysis* with proper points-to information [Ste96, SH97], and with global context information on compiler-built-in functions and standard libraries, the amount of manual annotations can be reduced significantly. First results from improved research prototypes indicate that the number of manual annotations can at least be halved to get equivalent results to the algorithm in its current version.
- Results of the *Cohesion Analysis* are inherently valid for one specific system configuration. This fact mainly results from late-bound function pointers—typically configured after compile-time—and from configurable modes of operation of an automotive system. The current methodology, as pointed out in Chapter 4, relies on external expertise to identify a representative configuration. To overcome this weakness, the *Cohesion Analysis* will be extended to cope with multiple weighted component graphs, to calculate a more general decomposition of the *AUTOSAR Basic Software*.
- To support timing-aware composition for *AUTOSAR Basic Software*, contracts for middleware components can be extended by timing contracts that contain information on call dependencies as much as parametrized approximation formulas for all operations [10]. In consequence, end-to-end timing for *AUTOSAR* application components can

be calculated (or at least be bound) at model level. *AUTOSAR* in its current version lacks of standardized mechanisms for timing-aware software composition within its *Basic Software*. Hence, the concept of timing-aware composition can be used to improve *AUTOSAR* by aspects of execution-times at system level.

6.4 Conclusion

Within this thesis, the concept of component based middleware was introduced to the Automotive Open System Architecture (*AUTOSAR*) via the *COMPASS* methodology. Since this automotive system standard so far is based on layered communication middleware, a new component based design has been defined. However, to keep in line with the standard, the external interfaces of the communication middleware completely resemble the external interfaces of the conventional version.

A component model for the component middleware was specified, and in addition, middleware component classes as much as class variants of different size and complexity have been identified as building blocks for the new type of middleware. The component class identification was achieved via static source code analysis of an industrial implementation of traditional *AUTOSAR* middleware. The proposed *Cohesion Analysis* utilizes control-flow information, and information on data field usage. Its complexity is linear to the size of the analyzed source code, hence it is applicable to complex real-world systems.

To foster automatic generation of application specific and hence custom-tailored resource-aware middleware, a model driven process was defined, which is capable of synthesizing component based middleware from prefabricated middleware components, architectural templates, and application models. The specified *Connector Transformation* generates platform specific models from platform independent application models by transforming explicit connectors within the PIMs into platform specific middleware component architectures and generated interface adapter components within the PSMs. In a subsequent step the PSMs are transformed into application and node specific executables.

Experimental results showed the capability of the provided methodology to reduce the middleware's memory foot-print by up to 30% while CPU usage was reduced by up to 10%.

Therefore, this thesis can finally be concluded by answering the two research questions issued in Chapter 1:

1. Yes. It is possible to apply the component paradigm in *AUTOSAR* not only at application level, but also to the component middleware, especially to the communication subsystem.

2. Yes. Component based, application specific, and hence custom-tailored *AUTOSAR* middleware can automatically be synthesized from application models and prefabricated middleware components.

Appendix A

SPEM

The *Software Process Engineering Metamodel* (*SPEM*), as defined by the OMG in [OMG05], allows the specification of complex software engineering processes in a model driven way. Process models not only help in project optimization, but also provide valuable information on resource usage, workflow dependencies, and tool chain requirements. In addition, a well-defined process model fosters the integration of specified engineering processes into other, more complex and even distributed ones. As this feature plays an important role within the automotive industry [PBKS07], it is not surprising that *SPEM* comes to use within the domain of *AUTOSAR*.

Software processes in *SPEM* are expressed in UML 2 diagrams. To increase the models' readability, *SPEM* provides its own UML profile, which defines a set of stereotypes and corresponding graphical syntax. Figure A.1 depicts some that are frequently used within the *AUTOSAR* methodology specification.

1. **«WorkProduct»**: Within the context of *SPEM*, a work product is anything (a model, source code, etc.) produced, consumed, or modified by a process. The stereotype «WorkProduct» describes a class of work product within a process, and is mostly used for artifacts that are neither documents (source code, executables, configuration files, etc.) nor UML 2.0 models. All work products («WorkProduct», «UMLModel» and «Document») may be altered by an activity, and thus may exist in multiple distinct states across one process model. To distinguish different states, they are directly annotated at the affected work product.
2. **«Document» and «UMLModel»**: Both stereotypes describe a specialization of a work product, and are added to the UML profile for

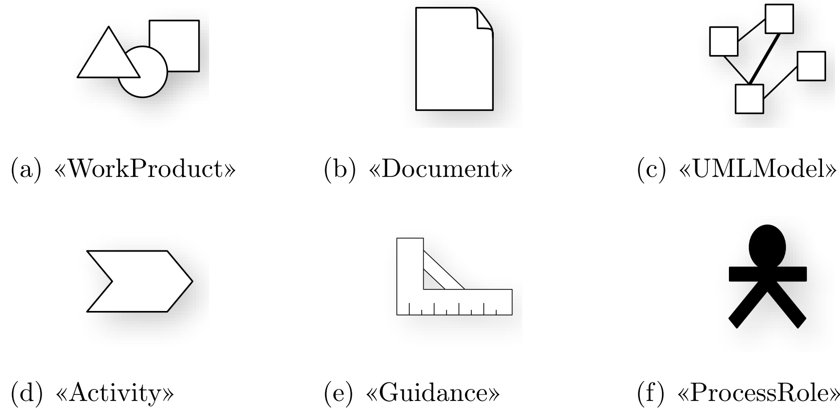

(a) «WorkProduct» (b) «Document» (c) «UMLModel»
(d) «Activity» (e) «Guidance» (f) «ProcessRole»

Figure A.1: SPEM stereotypes

reasons of comprehensibility. The «Document» stereotype typically denotes source code, compiled binaries, configuration files, and control files for the tool chain, while the «UMLModel» stereotype denotes UML models, or parts of them.

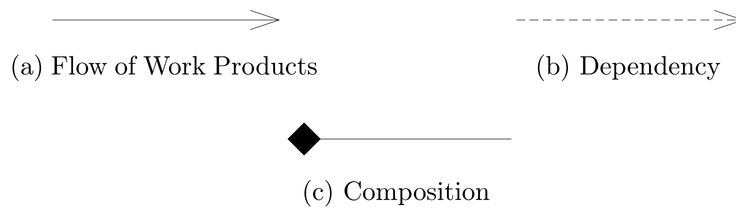
3. **«Activity»**: This stereotype describes a piece of work executed by one process role: tasks, operations, and actions. An activity is associated with inputs and outputs that are work products. It is performed by one actor, a «ProcessRole», but may be assisted by multiple others.

4. **«Guidance»**: The «Guidance» stereotype may be associated with arbitrary model elements to provide additional information. It is for example used to assign tools to activities.

5. **«ProcessRole»**: This stereotype defines responsibilities for work products. It denotes active resources that execute specific activities, or that at least assist in execution. Model elements of stereotype «ProcessRole» are performers of activities within *SPEM* process models.

Besides these model elements, *SPEM* defines a set of relations between them. The ones that are used here are depicted in Figure A.2:

1. **Flow of work product**: The flow of a work product represents the relation between work products and activities. It is graphically represented by a line with an arrowhead (see Figure A.2a), originating from its source and targeted at its destination. Flows of work products are used to identify inputs and outputs of activities.



(a) Flow of Work Products (b) Dependency

(c) Composition

Figure A.2: SPEM associations

2. **Dependency**: Dependencies represent unidirectional relations between work products, and define who depends on whom. They are represented by a dotted line with an arrowhead, originating in the model element depending on the one, at which the arrowhead points (see Figure A.2b).

3. **Composition**: A composition denotes that one work product consists of (or contains) other work products. In general, it is meant to show a part-total relation. Thus, it is graphically represented by a line with a solid diamond at the total's end (see Figure A.2c).

As *SPEM* is a UML profile, all types of UML diagrams (class diagrams, use-case diagrams, activity diagrams, etc.) can be used to outline different aspects of the modeled process.

Appendix B

Used UML 2 Diagram Types

The following list outlines the basic types of UML 2 diagrams that are used within the *COMPASS* methodology.

1. **Component Diagram**: This type of diagram is used to specify software architectures—the software's functional decomposition. Component diagrams are typically used in conjunction with deployment diagrams. They mainly consist of component artifacts and connectors that link the components' provided and required interfaces (see Chapter 2). The *COMPASS* methodology utilizes component diagrams to specify the application at the platform independent level within PIMs, but also at platform specific level to capture the results of the PIM-to-PSM transformation (see Chapter 5).

2. **Composite Structure Diagram**: Diagrams of this type define a component's internal structure and its points of interaction in terms of so called parts and ports (respectively connectors between ports). Parts represent a subset of possible artifacts contained within a component. Hence, a part can be used to represent a component class. Composite structure diagrams provide a white-box view of a component, and thus are used to specify a component's internal architecture. They are well suited to specify architectural patterns. The main application of composite structure diagrams in the context of this thesis is to specify architectural patterns of connector fragments (also referred to as connector templates), used by the PIM-to-PSM transformation to synthesize communication middleware.

3. **Deployment Diagram**: To describe the system's physical structure, especially the relevant ECUs and the connecting bus systems, and to

map the application components to the specific ECUs, deployment diagrams come to use. The main model artifacts used within deployment diagrams are system building blocks, like ECUs and bus systems, and components. Diagrams of this type are used as input for the PIM-to-PSM transformation (the *Connector Transformation*), as they provide information about the application components' physical location, and in consequence information about the types of explicit connectors.

All of these diagrams are additionally augmented with various contracts and tags, to increase precision and comprehensibility.

Appendix C

FlexRay Communication Layers

Table C.1 provides full information on the classification of all operations within the *FlexRay Driver* layer, the *FlexRay Interface* layer, and the *FlexRay Transceiver Driver* layer with respect to *AUTOSAR* R2, as much as their mapping onto identified middleware components.

The *COMPASS* communication middleware is built upon following communication primitives: *Transceiver Driver* (TD), *Time Service absolute* (Time Service a), *Time Service relative* (Time Service r), *Time Service global* (Time Service g), *Time Service full* (Time Service full), *Status* (S), *Media Test Symbol* (MTS), *WakeUp Pattern* (WUP), *Communication Base* (Base S), *Communication Base without cluster functionality* (Base N), *Transmitter* (T), and *Receiver* (R).

Each operation is represented by one row, while the columns represent the *COMPASS* middleware component classes. If an operation has to be implemented by a middleware component, a mark is placed in the operation's row at the component's column.

	TD	Time Service				S	MTS	WUP	Base		T	R
Operation		a	r	g	full				S	N		
FrIf_GetVersionInfo									x	x		
FrIf_Init									x			
FrIf_ControllerInit									x	x		
FrIf_ClusterInit									x			
FrIf_Transmit											x	
FrIf_StartControllerCommunication									x	x		
FrIf_StartClusterCommunication									x			
FrIf_HaltControllerCommunication									x	x		
FrIf_HaltClusterCommunication									x			
FrIf_AbortControllerCommunication									x	x		
FrIf_AbortClusterCommunication									x			
FrIf_SetControllerState									x	x		
FrIf_SetClusterState									x			
FrIf_GetControllerState									x	x		
FrIf_GetClusterState									x			
FrIf_SetWakeupChannel								x				
FrIf_SendWUP								x				

continued on next page

Operation	TD	Time Service				S	MTS	WUP	Base		T	R
		a	r	g	full				S	N		
FrIf_SendMTS							x					
FrIf_CheckMTS							x					
FrIf_GetSyncState						x						
FrIf_SetExtControllerSync				x	x							
FrIf_SetExtClusterSync				x	x							
FrIf_GetPOCStatus						x						
FrIf_GetGlobalTime				x	x							
FrIf_GetMactroticksPerCycle				x	x							
FrIf_ConvertNanosecToMacroticks				x	x							
FrIf_ConvertMacroticksToNanosec				x	x							
FrIf_SetAbsoluteTimer		x			x							
FrIf_SetRelativeTimer			x		x							
FrIf_CancelAbsoluteTimer		x			x							
FrIf_CancelRelativeTimer			x		x							
FrIf_EnableAbsoluteTimerIRQ		x			x							
FrIf_EnableRelativeTimerIRQ			x		x							
FrIf_AckAbsoluteTimerIRQ		x			x							
FrIf_AckRelativeTimerIRQ			x		x							
FrIf_DisableAbsoluteTimerIRQ		x			x							
FrIf_DisableRelativeTimerIRQ			x		x							
FrIf_SetTransceiverMode	x											
FrIf_SetClusterTransceiverMode	x											
FrIf_GetTransceiverMode	x											
FrIf_GetTransceiverWUReason	x											
FrIf_EnableTransceiverWakeup	x											
FrIf_EnableClusterTransceiverWakeup	x											
FrIf_DisableTransceiverWakeup	x											
FrIf_DisableClusterTransceiverWakeup	x											
FrIf_ClearTransceiverWakeup	x											
FrIf_ClearClusterTransceiverWakeup	x											
FrIf_JoblistExec_<ClstIdx>									x	x		
FrIf_MainFunction_<ClstIdx>									x	x		
<User>_RxIndication												x
<User>_TxConfirmation											x	
<User>_TriggerTransmit											x	
Fr_Init									x	x		
Fr_ControllerInit									x	x		
Fr_SendMTS							x					
Fr_CheckMTS							x					
Fr_StartCommunication									x	x		
Fr_HaltCommunication									x	x		
Fr_AbortCommunication									x	x		
Fr_SendWup								x				
Fr_SetWakeupChannel								x				
Fr_SetExtSync				x	x							
Fr_GetSyncState						x						
Fr_GetPOCStatus						x			x	x		
Fr_TransmitTxLSdu											x	
Fr_ReceiveRxLSdu												x
Fr_CheckTxLSduStatus											x	
Fr_ConfigAllBuffers									x	x		
Fr_GetGlobalTime				x	x				x	x		
Fr_SetAbsoluteTimer		x			x				x	x		
Fr_SetRelativeTimer			x		x							
Fr_CancelAbsoluteTimer		x			x							
Fr_CancelRelativeTimer			x		x							
Fr_EnableAbsoluteTimerIRQ		x			x				x	x		
Fr_EnableRelativeTimerIRQ			x		x							
Fr_AckAbsoluteTimerIRQ		x			x				x	x		
Fr_AckRelativeTimerIRQ			x		x							
Fr_DisableAbsoluteTimerIRQ		x			x							
Fr_DisableRelativeTimerIRQ			x		x							
Fr_GetVersionInfo									x	x		
FrTrcv_TrcvInit	x											
FrTrcv_SetTransceiverMode	x											
FrTrcv_GetTransceiverMode	x											
FrTrcv_GetTransceiverWUReason	x											
FrTrcv_GetVersionInfo	x											
FrTrcv_DisableTransceiverWakeup	x											
FrTrcv_EnableTransceiverWakeup	x											
FrTrcv_ClearTransceiverWakeup	x											
FrTrcv_MainFunctiorn	x											

continued on next page

Operation	TD	Time Service				S	MTS	WUP	Base		T	R
		a	r	g	full				S	N		
FrTrcv_Cbk_WakeupByTransceiver	x											

Table C.1: FlexRay communication primitives

Appendix D

Manual Annotations for Component Recognition

The component recognition algorithm (*Cohesion Analysis*) described in Chapter 4 requires manual identification of first-class functions that are to be considered by the analysis, and a specification of abstract memory locations (structure fields) that relate to component classes. The following sections denote the functions and structure fields as marked by domain experts, to gain the results presented within this thesis.

D.1 Functions

Table D.1 contains all functions of the analyzed communication stack, which should be included within middleware components, and have to be contained within a provided-interface.

Function Name
FrIf_GetVersionInfo
FrIf_Init
FrIf_ControllerInit
FrIf_ClusterInit
FrIf_Transmit
FrIf_StartControllerCommunication
FrIf_StartClusterCommunication
FrIf_HaltControllerCommunication
FrIf_HaltClusterCommunication

continued on next page

Function Name
FrIf_AbortControllerCommunication
FrIf_AbortClusterCommunication
FrIf_SetControllerState
FrIf_SetClusterState
FrIf_GetControllerState
FrIf_GetClusterState
FrIf_SetWakeupChannel
FrIf_SendWUP
FrIf_SendMTS
FrIf_CheckMTS
FrIf_GetSyncState
FrIf_SetExtControllerSync
FrIf_SetExtClusterSync
FrIf_GetPOCStatus
FrIf_GetGlobalTime
FrIf_GetMactroticksPerCycle
FrIf_ConvertNanosecToMacroticks
FrIf_ConvertMacroticksToNanosec
FrIf_SetAbsoluteTimer
FrIf_SetRelativeTimer
FrIf_CancelAbsoluteTimer
FrIf_CancelRelativeTimer
FrIf_EnableAbsoluteTimerIRQ
FrIf_EnableRelativeTimerIRQ
FrIf_AckAbsoluteTimerIRQ
FrIf_AckRelativeTimerIRQ
FrIf_DisableAbsoluteTimerIRQ
FrIf_DisableRelativeTimerIRQ
FrIf_SetTransceiverMode
FrIf_SetClusterTransceiverMode
FrIf_GetTransceiverMode
FrIf_GetTransceiverWUReason
FrIf_EnableTransceiverWakeup
FrIf_EnableClusterTransceiverWakeup
FrIf_DisableTransceiverWakeup
FrIf_DisableClusterTransceiverWakeup

continued on next page

Function Name
FrIf_ClearTransceiverWakeup FrIf_ClearClusterTransceiverWakeup
FrIf_JoblistExec_<ClstIdx>
FrIf_MainFunction_<ClstIdx>
Fr_10_ERAY_Init Fr_10_ERAY_ControllerInit
Fr_10_ERAY_SendMTS Fr_10_ERAY_CheckMTS
Fr_10_ERAY_StartCommunication Fr_10_ERAY_HaltCommunication Fr_10_ERAY_AbortCommunication
Fr_10_ERAY_SendWUP Fr_10_ERAY_SetWakeupChannel
Fr_10_ERAY_SetExtSync Fr_10_ERAY_GetSyncState Fr_10_ERAY_GetPOCStatus
Fr_10_ERAY_TransmitTxLSdu Fr_10_ERAY_ReceiveRxLSdu Fr_10_ERAY_CheckTxLSduStatus
Fr_10_ERAY_ConfigAllBuffers
Fr_10_ERAY_GetGlobalTime Fr_10_ERAY_SetAbsoluteTimer Fr_10_ERAY_SetRelativeTimer Fr_10_ERAY_CancelAbsoluteTimer Fr_10_ERAY_CancelRelativeTimer Fr_10_ERAY_EnableAbsoluteTimerIRQ Fr_10_ERAY_EnableRelativeTimerIRQ Fr_10_ERAY_AckAbsoluteTimerIRQ Fr_10_ERAY_AckRelativeTimerIRQ Fr_10_ERAY_DisableAbsoluteTimerIRQ Fr_10_ERAY_DisableRelativeTimerIRQ
Fr_10_ERAY_GetVersionInfo
FrTrcv_TrcvInit
FrTrcv_SetTransceiverMode FrTrcv_GetTransceiverMode
FrTrcv_GetTransceiverWUReason

continued on next page

Function Name
FrTrcv_GetVersionInfo
FrTrcv_DisableTransceiverWakeup FrTrcv_EnableTransceiverWakeup FrTrcv_ClearTransceiverWakeup
FrTrcv_MainFunctiorn
FrTrcv_Cbk_WakeupByTransceiver

Table D.1: Marked functions

D.2 Structure Fields

Table D.2 contains all structure fields that have been marked as irrelevant for the component recognition algorithm. Fields are specified by the data structure's type and by the field's name. An asterisk (*) at the field's name represents a wild-card, denoting all fields of the specific data structure.

Stucture Type	**Field Name**
FrIf_TrcvFunctionType	*
FrIf_ClusterType	*
FrIf_CtrlType	*
FrIf_RootConfigType	*
FrIf_IdxMappingType	DriverNumber
FrIf_EntityCountersType	NrClst
FrIf_EntityCountersType	*
FrIf_IdxMappingType	FrIdx
Fr_10_ERAY_CtrlCfgType	BufferCfgPtr
Fr_10_ERAY_BufferCfgType	*
Fr_10_ERAY_CfgType	*
Fr_10_ERAY_CtrlCfgType	nBufferCfgSize

Table D.2: Marked fields

List of Figures

2.1 *UML 2* notation of a component 14
2.2 *UML* connector notations 16
2.3 Virtual function bus (VFB) 18
2.4 Model transformation 24
2.5 Guided transformation 26

3.1 ECU design 30
3.2 AUTOSAR software architecture [AUT08a] 32
3.3 Simplified FlexRay communication stack 34
3.4 Software component and runnable entities 37
3.5 Overview of the AUTOSAR methodology [AUT08b] 41

4.1 Component based middleware for *AUTOSAR* 44
4.2 Meta-model of COMPASS data types (simplified) 46
4.3 Meta-model of COMPASS components 48
4.4 Middleware building blocks 50
4.5 Meta-model of component interfaces 52
4.6 Source code example and AST 63

5.1 PIM to PSM transformation of connector artifacts 71
5.2 Meta-model of *COMPASS* explicit connectors 73
5.3 Notation of contracts in UML 75
5.4 Development process for middleware components 77
5.5 Connector Transformation 78
5.6 Sender-Receiver connector template 81
5.7 Client-Server connector template 83

6.1 Recognized middleware components for *AUTOSAR* 95
6.2 Automotive door-lock application 97
6.3 Result of applied model transformation. 100

A.1 SPEM stereotypes 110

A.2 SPEM associations 111

List of Algorithms

1 Cohesion Analysis 62
2 Connector Transformation 88

List of Tables

5.1 Connector classification by interaction type 72

6.1 Base characteristics of the analyzed communication stack . . . 93
6.2 Code characteristics of the analyzed communication stack . . . 94
6.3 BSW components for FlexRay 101

C.1 FlexRay communication primitives 116

D.1 Marked functions 120
D.2 Marked fields 120

Bibliography

[ASSK02] Aditya Agrawal, Tihamer Levendovszkyand Jon Sprinkle, Feng Shi, and Gabor Karsai. Generative Programming via Graph Transformations in the Model-Driven Architecture. In *OOPSLA 2002 Workshop in Generative Techniques in the context of Model Driven Architecture*, OOPSLA, 2002. Available from: http://www.softmetaware.com/oopsla2002/karsaig.pdf, last visited on 10.04.2009.

[ATK92] A. L. Ananda, B. H. Tay, and E. K. Koh. A Survey of Asynchronous Remote Procedure Calls. *ACM SIGOPS Operating Systems Review*, 26(2):92–109, April 1992, ACM SIGOPS.

[ATL06] ATLAS Group. *ATL: Atlas Transformation Language*. LINA and INRIA, February 2006. Available from: http://www.eclipse.org/m2m/atl/doc/ATL_User_Manual%5Bv0.7%5D.pdf, last visited on 10.01.2009.

[AUT06a] AUTOSAR GbR. *Specification of FlexRay Driver V2.0.0*, 2006. Available from: http://www.autosar.org/download/AUTOSAR_SWS_FlexRay_Driver.pdf, last visited on 23.11.2007.

[AUT06b] AUTOSAR GbR. *Specification of FlexRay Interface V2.0.0*, 2006. Available from: http://www.autosar.org/download/AUTOSAR_SWS_FlexRay_Interface.pdf, last visited on 23.11.2007.

[AUT06c] AUTOSAR GbR. *Specification of FlexRay Transceiver Driver V1.0.1*, 2006. Available from: http://www.autosar.org/download/AUTOSAR_SWS_FlexRayTransceiver.pdf, last visited on 23.11.2007.

[AUT08a] AUTOSAR GbR. *Layered Software Architecture V2.2.1 R3.0*, 2008. Available from: http://www.autosar.org/download/AUTOSAR_LayeredSoftwareArchitecture.pdf, last visited on 11.1.2009.

[AUT08b] AUTOSAR GbR. *Methodology V1.2.1 R3.0*, 2008. Available from: http://www.autosar.org/download/AUTOSAR_Methodology.pdf, last visited on 10.01.2009.

[AUT08c] AUTOSAR GbR. *Software Component Template V3.0.1 R3.0*, 2008. Available from: http://www.autosar.org/download/AUTOSAR_SoftwareComponentTemplate.pdf, last visited on 12.1.2009.

[AUT08d] AUTOSAR GbR. *Specification of FlexRay Driver V2.2.2 R3.1*, 2008. Available from: http://www.autosar.org/download/specs_aktuell/AUTOSAR_SWS_FlexRayDriver.pdf, last visited on 26.01.2009.

[AUT08e] AUTOSAR GbR. *Specification of FlexRay Interface V3.0.3 R3.1*, 2008. Available from: http://www.autosar.org/download/specs_aktuell/AUTOSAR_SWS_FlexRayInterface.pdf, last visited on 26.01.2009.

[AUT08f] AUTOSAR GbR. *Specification of FlexRay Transceiver Driver V1.2.3*, 2008. Available from: http://www.autosar.org/download/specs_aktuell/AUTOSAR_SWS_FlexRayTransceiver.pdf, last visited on 26.01.2009.

[AUT08g] AUTOSAR GbR. *Specification of RTE V2.0.1 R3.0*, 2008. Available from: http://www.autosar.org/download/AUTOSAR_SWS_RTE.pdf, last visited on 11.1.2009.

[Bál02] Dusan Bálek. *Connectors in Software Architectures.* PhD thesis, Charles University Prague, Faculty of Mathematics and Physics; Department of Software Engineering, Prague, Feb. 2002. Available from: http://nenya.ms.mff.cuni.cz/publications/balek_phd.pdf, last visited on 31.01.2009.

[BALL89] Brian Bershad, Thomas Anderson, Edward Lazowska, and Henry Levy. Lightweight Remote Procedure Call. In *Proceedings of the 12th ACM symposium on Operating systems principles*, pages 102–113, New York, NY, USA, 1989. ACM SIGOPS.

[BCS04] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. The fractal component model [online]. February 2004. Available from: http://fractal.objectweb.org/specification/index.html, last visited on 10.02.2009.

[BDH+98] Manfred Broy, Anton Deimel, Juergen Henn, Kai Koskimies, Frantisek Plasil, Gustav Pomberger, Wolfgang Pree, Michael Stal, and Clemens Szyperski. What Characterizes a (Software) Component?". *Software - Concepts and Tools*, 19(1):49–56, February 1998, Springer-Verlag, Heidelberg.

[BN84] Andrew Birrell and Bruce Nelson. Implementing Remote Procedure Calls. In *ACM Transactions on Computer Systems (TOCS)*, volume 2, New York, NY, USA, 1984. ACM.

[BP01] Dusan Bálek and Frantisek Plasil. Software Connectors and their Role in Component Deployment. In *Proceedings of the IFIP TC6/WG6.1 Third International Working Conference on New Developments in Distributed Applications and Interoperable Systems*, volume 198 of *IFIP Conference Proceedings*, pages 69–84, Deventer, The Netherlands, 2001. Kluwer, B.V.

[BP04] Tomas Bures and Frantisek Plasil. Communication Style Driven Connector Configurations. In *Proceedings of the 1st International Conference on Software Engineering Research and Applications (SERA 2003), Selected Revised Papers*, volume 3026/2004 of *Lecture Notes in Computer Science*, pages 102–116. Springer Berlin / Heidelberg, 2004.

[Bro98] Manfred Broy. A Uniform Mathematical Concept of a Component. *Software - Concepts and Tools*, 19(1):57–59, February 1998, Springer-Verlag, Heidelberg.

[CH01] Bill Councill and George T. Heineman. Component-based software engineering. In Heineman and Councill [HC01], chapter Definition of a Software Component and its Elements, pages 5–19.

[CHJK02] Ivica Crncovic, Brahim Hnich, Torsten Jonsson, and Zeynep Kiziltan. Building Reliable Component-Based Software Systems. In Crnkovic and Larsson [CL02], chapter Basic Concepts in CBSE, pages 3–22.

[CL02] Ivica Crnkovic and Magnus Larsson, editors. *Building Reliable Component-Based Software Systems.* Artech House, Norwood, MA, 1. publ. edition, ISBN 1-580-53327-2, 2002.

[COM07] COMPASS. Component Based Automotive System Software [online]. 2005–2007. Available from: `http://www.infosys.tuwien.ac.at/compass`, last visited on 24.01.2009.

[DMM98] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-Based Alias Analysis. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI)*, pages 106–117, New York, NY, USA, June 1998. ACM.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-Sensitive Interprocedural Points-to Analysis in the Presence of

Function Pointers. *ACM SIGPLAN Notices*, 29(6):242–256, June 1994, ACM.

[FHHW03] T. Führer, F. Hartwich, R. Hugel, and H. Weiler. FlexRay – The Communication System for Future Control Systems in Vehicles; Document Number 2003-01-0110. In *In-Vehicle Networks, Safety Critical Systems, Accelerated Testing, and Reliability 2003 (SP-1783)*, SAE Special Publication Papers, Detroit, MI, USA, March 2003. Society of Automotive Engineers.

[GSL+03] Aniruddha S. Gokhale, Douglas C. Schmidt, Tao Lu, Balachandran Natarajan, and Nanbor Wang. CoSMIC: An MDA Generative Tool for Distributed Real-time and Embedded Applications. In *International Middleware Conference, Workshop Proceedings of Middleware Workshops*, pages 300–306. PUC-Rio, 2003.

[Han05] P. Hansen. New S-Class Mercedes: Pioneering Electronics. *The Hansen Report on Automotive Electronics*, 18(8):1–2, October 2005, Paul Hansen. Available from: `http://www.hansenreport.com/search_Detail.cfm?ProductID=229`, last visited on 06.02.2009.

[HBCC99] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural Pointer Alias Analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4):848–894, July 1999, ACM.

[HC01] George T. Heineman and William T. Councill, editors. *Component-Based Software Engineering*. Addison-Wesley, 1st edition, ISBN 0-201-70485-4, 2001.

[HSF+04] H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, J.-L. Mate, K. Nishikawa, and T. Scharnhorst. AUTomotive Open System ARchitecture — An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E-Architectures. In *Proceedings of the Convergence International Congress & Exposition On Transportation Electronics*, SAE-2004-21-0042, Detroit, MI, USA, October 2004.

[ICG07] Valerie Issarny, Mauro Caporuscio, and Nikolaos Georgantas. A Perspective on the Future of Middleware-based Software Engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 244–258, Washington, DC, USA, May 2007. IEEE Computer Society.

[IEE00] IEEE Standards Board. *IEEE Recommended Practice for Architectural Description of Software-intensive Systems—Std. 1471-2000.*

IEEE Computer Society, New York, NY, USA, ISBN 0-7381-2518-0, 2000.

[ISO99] ISO (International Organization for Standardization), 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland. *Road Vehicles – Diagnostic Systems – Keyword Protocol 2000 – Part 3: Application Layer*, 1. edition, 1999. Available from: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=23921, last visited on 12.1.2009.

[ISO03a] ISO (International Organization for Standardization), 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland. *Road Vehicles – Controller Area Network (CAN) – Part 1: Data Link Layer and Physical Signalling*, 1. edition, 2003. Available from: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=33422, last visited on 11.1.2009.

[ISO03b] ISO (International Organization for Standardization), 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland. *Road Vehicles – Controller Area Network (CAN) – Part 2: High-Speed Medium Access Unit*, 1. edition, 2003. Available from: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=33423, last visited on 11.1.2009.

[ISO04] ISO (International Organization for Standardization), 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland. *Road Vehicles – Diagnostics on Controller Area Networks (CAN) – Part 2: Network Layer Services*, 1. edition, April 2004. Available from: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=33616, last visited on 12.1.2009.

[ISO06] ISO (International Organization for Standardization), 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland. *Road Vehicles – Unified Diagnostic Services (UDS) – Part 1: Specification and Requirements*, 1. edition, 2006. Available from: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=45293, last visited on 12.1.2009.

[JK06] Frédéric Jouault and Ivan Kurtev. On the Architectural Alignment of ATL and QVT. In *Proceedings of the 2006 ACM symposium*

on Applied computing (SAC06), pages 1188–1195, New York, NY, USA, April 2006. ACM Press.

[Kel07] Stephen Kell. Rethinking Software Connectors. In *SYANCO '07: International workshop on Synthesis and analysis of component connectors*, pages 1–12, New York, NY, USA, 2007. ACM.

[KG99] A. Kelkar and R. Gamble. Understanding the Architectural Characteristics behind Middleware Choices. In *Proceeding of the 1st International Conference in Information Reuse and Integration*, November 1999. Available from: `http://www.seat.utulsa.edu/papers/KG99.pdf`, last visited on 05.01.2008.

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture—Practice and Promise*. Addison-Wesley, 1. print. edition, ISBN 0-321-19442-X, 2003.

[LIN06] LIN Consortium. *LIN Specification Package*, November 2006. Available from: `http://www.lin-subbus.de/index.php?pid=9&lang=en`, last visited on 11.1.2009.

[LQS05] Matthieu Leclercq, Vivien Quema, and Jean-Bernard Stefani. DREAM: A Component Framework for Constructing Resource-Aware, Configurable Middleware. *IEEE Distributed Systems Online*, 6(9):1, 2005, IEEE Educational Activities Department. ISSN 1541-4922.

[LSK+01] Jong Kook Lee, Seung Jae Seung, Soo Dong Kim, Woo Hyun, and Dong Han Han. Component Identification Method with Coupling and Cohesion. In *Proceedings of the Eight Asia-Pacific Software Engineering Conference 2001 (APSEC 2001)*, pages 79–86, Washington, DC, USA, Dezember 2001. IEEE Computer Society.

[LW05] Kung-Kiu Lau and Zheng Wang. A Taxonomy of Software Component Models. In *Proceedings of the 31st Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA)*, pages 88–95, August 2005.

[Mey92a] Bertrand Meyer. Advances in object-oriented software engineering. In Meyer and Mandrioli [MM92], chapter Design by Contract, pages 1–50.

[Mey92b] Bertrand Meyer. Applying 'design by contract'. *IEEE Computer*, 25(10):40–51, October 1992, IEEE Computer Society.

[Mey03] Bertrand Meyer. The Grand Challenge of Trusted Components. In *Proceedings of the 25th International Conference on Software Engineering (ICSE03)*, pages 660–667. IEEE Computer Society, May 2003.

[MHB+01] R. Mores, G. Hay, R. Belschner, J. Berwanger, C. Ebner, S. Fluhrer, E. Fuchs, B. Hedenetz, W. Kuffner, A. Krüger, P. Lohrmann, D. Millinger, M. Peller, J. Ruh, A. Schedl, and M. Sprachmann. FlexRay – The Communication System for Advanced Automotive Control Systems; Document Number 2001-01-0676. In *SAE Transactions 2001*, volume 110, pages 303–314, Detroit, MI, USA, March 2001. Society of Automotive Engineers.

[Mic09] Microsoft. *COM (Component Object Model)*. Microsoft, 2009. Available from: `http://msdn.microsoft.com/en-us/library/ms680573.aspx`, last visited on 31.01.2009.

[MM92] Bertrand Meyer and Dino Mandrioli, editors. *Advances in Object-Oriented Software Engineering*. Prentice-Hall object-oriented series. Prentice Hall, Englewood Cliffs, NJ, ISBN 0-13-006578-1, 1992.

[MSM06] Sam Malek, Chiyoung Seo, and Nenad Medvidovic. Tailoring an Architectural Middleware Platform to a Heterogeneous Embedded Environment. In *Proceedings of the 6th International Workshop on Software Engineering and Middleware (SEM06)*, pages 63–70, New York, NY, USA, 2006. ACM Press.

[NB02] Eivind J. Nordby and Martin Blom. Building Reliable Component-Based Software Systems. In Crnkovic and Larsson [CL02], chapter Semantic Integrity in CBD, pages 115–133.

[NT95] Oscar Nierstrasz and Dennis Tsichritzis, editors. *Object-Oriented Software Composition*. Prentice-Hall Object-Oriented Series. Prentice-Hall, 1st edition, ISBN 0-13-220674-9, December 1995.

[OIC08] OICA. International Organization of Motor Vehicle Manufacturers, 2007 Production Statistics [online]. 2008. Available from: `http://oica.net/category/production-statistics/2007-statistics/`, last visited on 04.01.2009.

[OMG03a] OMG. *MDA Guide Version 1.0.1*, 2003. Available from: `http://www.omg.org/docs/omg/03-06-01.pdf`, last visited on 04.01.2009.

[OMG03b] OMG. *Meta Object Facility (MOF) 2.0 Core Specification Version 2.0*, 2003. Available from: `http://www.omg.org/cgi-bin/apps/doc?ptc/03-10-04.pdf`, last visited on 06.01.2009.

[OMG05] OMG. *Software Process Engineering Metamodel Specification*, 2005. Available from: `http://www.omg.org/docs/formal/05-01-06.pdf`, last visited on 10.01.2009.

[OMG06] OMG. *CORBA Component Model Specification Version 4.0*, 2006. Available from: `http://www.omg.org/docs/formal/06-04-01.pdf`, last visited on 04.01.2009.

[OMG07a] OMG. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*, 2007. Available from: `http://www.omg.org/cgi-bin/apps/doc?ptc/07-07-07.pdf`, last visited on 06.01.2009.

[OMG07b] OMG. *Unified Modeling Language: Superstructure*, 2007. Available from: `http://www.omg.org/cgi-bin/apps/doc?formal/07-02-03.pdf`, last visited on 05.01.2009.

[Pas02] Alessandro Pasetti. *Software Frameworks and Embedded Control Systems*, volume 2231/2002 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, ISBN 978-3-540-43189-3, 2002.

[PBKS07] Alexander Pretschner, Manfred Broy, Ingolf H. Kruger, and Thomas Stauner. Software Engineering for Automotive Systems: A Roadmap. In *FOSE '07: 2007 Future of Software Engineering*, pages 55–71, Washington, DC, USA, May 2007. IEEE, IEEE Computer Society.

[PCCB00] N. Parlavantzas, G. Coulson, M. Clarke, and G. Blair. Towards a Reflective Component Based Middleware Architecture. In Walter Cazzola, Shigeru Chiba, and Thomas Ledoux, editors, *On-Line Proceedings of ECOOP'2000 Workshop on Reflection and Metalevel Architectures*, 2000. Available from: `http://www.disi.unige.it/person/CazzolaW/ewrma2000-proceedings.html`, last visited on 05.01.2008.

[PVB99] Frantisek Plasil, Stanislav Visnovsky, and Miloslav Besta. Bounding Component Behavior via Protocols. In *Technology of Object-Oriented Languages and Systems, 1999. TOOLS 30. Proceedings*, pages 387–398, Santa Barbara, CA, USA, March 1999. IEEE Computer Society.

[RMRR98] Jason Robbins, Nenad Medvidovic, David Redmiles, and David Rosenblum. Integrating Architecture Description Languages with a Standard Design Method. In *Proceedings of the 20th International Conference on Software Engineering (ICSE98)*, pages 209–218. IEEE Computer Society Press, April 1998.

[RS02] Ralf H. Reussner and Heinz W. Schmidt. Using Parameterised Contracts to Predict Properties of Component Based Software Architectures. In Ivica Crnkovic, Stig Larsson, and Judith Stafford, editors, *Proceedings of the Workshop on Component-based Software Engineering at the 9th IEEE Conference and Workshops on Engineering of Computer-Based Systems*, Lund, Sweden, April 2002. Available from: `http://www.idt.mdh.se/~icc/cbse-ecbs2002/CBSE_ECBS2002-Proceedings.pdf`, last visited on 6.2.2009.

[RSB+04] Günther Rackl, Ulrike Sommer, Klaus Beschorner, Heinz Kössler, and Adam Bien. Komponentenbasierte Entwicklung auf Basis der "Model Driven Architecture". *ObjectSpectrum*, (5), 2004. Available from: `http://www.sigs.de/publications/os/2004/05/bien_beschorner_OS_05_04.pdf`, last visited on 27.07.2009.

[Sch02] Douglas C. Schmidt. Middleware for real-time and embedded systems. *Communications of the ACM*, 45(6):43–48, 2002, ACM Press. ISSN 0001-0782.

[Sel99] Bran Selic. Protocols and ports: Reusable inter-object behavior patterns. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 1999)*, pages 332–339. IEEE Computer Society, May 1999.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall, Upper Saddle River, NJ, USA, ISBN 0-13-182957-2, 1996.

[SH97] Marc Shapiro and Susan Horwitz. Fast and Accurate Flow-insensitive Points-to Analysis. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, New York, NY, USA, 1997. ACM.

[Sha93] Mary Shaw. Procedure Calls Are the Assembly Language of Software Interconnections: Connectors Deserve First Class Status. In D. Lamb, editor, *Studies of Software Design, ICSE '93 Workshop; selected papers*, volume 1078 of *LNCS*, pages 17–32, Baltimore, Maryland, USA,, May 1993. Springer-Verlag, Berlin.

[SOF09] SOFA. Sofa - software appliances [online]. 2009. Available from: `http://nenya.ms.mff.cuni.cz/projects/sofa/tools/doc/compmodel.html`, last visited on 09.02.2009.

[Ste96] Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, New York, NY, USA, 1996. ACM.

[Szy99] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1. edition, ISBN 0-201-17888-5, 1999.

[vO98] Rob C. van Ommering. Koala, a Component Model for Consumer Electronics Product Software. In *Development and Evolution of Software Architectures for Product Families*, volume 1429/1998 of *Lecture Notes in Computer Science*, pages 76–86. Springer Berlin / Heidelberg, 1998.

[vO04] Rob C. van Ommering. *Building Product Populations with Software Components*. PhD thesis, University of Groningen, 2004. Available from: `http://dissertations.ub.rug.nl/FILES/faculties/science/2004/r.c.van.ommering/thesis.pdf`, last visited on 31.01.2009.

[vOvdLKM00] Rob C. van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, 33(3):78–85, March 2000, IEEE Computer Society.

[WS01] Rainer Weinreich and Johannes Sametinger. Component-based software engineering. In Heineman and Councill [HC01], chapter Component Models and Component Services: Concepts and Principles, pages 33–48.

Author's Publications

[1] Dietmar Schreiner, Markus Schordan, Gergö Barany, and Karl M. Göschka. Source Code Based Component Recognition in Software Stacks for Embedded Systems. In *Proceedings of the 2008 IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications (MESA08)*, pages 463–468. IEEE, October 2008.

[2] Thomas M. Galla, Dietmar Schreiner, Wolfgang Forster, Christof Kutschera, Karl M. Göschka, and Martin Horauer. Refactoring an Automotive Embedded Software Stack using the Component-Based Paradigm. In *Proceedings of the Second IEEE International Symposium on Industrial Embedded Systems (SIES 2007)*, IEEE, pages 200–208. IEEE, January 2007.

[3] Wolfgang Forster, Christof Kutschera, Dietmar Schreiner, and Karl M. Göschka. A Unified Benchmarking Process for Components in Automotive Embedded Systems Software. In *Proceedings of the 10th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2007)*, IEEE, pages 41–45. IEEE, April 2007.

[4] Dietmar Schreiner and Karl M. Göschka. A Component Model for the AUTOSAR Virtual Function Bus. In *Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC '07)*, volume 2, pages 635–641. IEEE, July 2007.

[5] Dietmar Schreiner and Karl M. Göschka. Building Component Based Software Connectors for Communication Middleware in Distributed Embedded Systems. In *Proceedings of the 2007 ASME/IEEE International Conference on Mechatronic and Embedded Systems and Applications (MESA07)*. ASME, September 2007. CD-ROM, ISBN 0-7918-3806-4.

[6] Dietmar Schreiner, Markus Schordan, and Karl M. Göschka. Component Based Middleware-Synthesis for AUTOSAR Basic Software. In *Proceedings of the 12th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2009)*, IEEE, pages 236–243. IEEE Computer Society, March 2009.

[7] Dietmar Schreiner and Karl M. Göschka. Modeling Component Based Embedded Systems Applications with Explicit Connectors in UML 2.0. In *Proceedings of the 2007 ACM symposium on Applied computing (SAC '07)*, pages 1494–1495, New York, NY, USA, March 2007. ACM Press.

[8] Dietmar Schreiner and Karl M. Göschka. Synthesizing Communication Middleware from Explicit Connectors in Component Based Distributed Architectures. In *Software Composition*, volume 4829/2007 of *LNCS*, pages 160–167. Springer, March 2007.

[9] Dietmar Schreiner and Karl M. Göschka. Explicit Connectors in Component Based Software Engineering for Distributed Embedded Systems. In *SOFSEM 2007: Theory and Practice of Computer Science, Proceedings*, volume 4362 of *LNCS*, pages 923–934. LNCS, Springer, January 2007.

[10] Dietmar Schreiner, Markus Schordan, and Jens Knoop. Adding Timing-Awareness to AUTOSAR Basic-Software – A Component Based Approach. In *Proceedings of the 12th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2009)*, pages 288–292. IEEE, IEEE Computer Society, March 2009.

Curriculum Vitae

Name: Dietmar Schreiner
Title: Dipl.-Ing.(FH)
Nationality: Austria
Date of Birth: 28.03.1972

Affiliation: Institute of Computer Languages
Languages and Compilers Group
Vienna University of Technology
Argentinierstr. 8/E185-1
A-1040 Vienna, Austria

e-Mail: schreiner@complang.tuwien.ac.at

Education

2005-2009	Doctoral Studies in Computer Science at Vienna University of Technology, Austria
2005	Graduation as "Dipl.-Ing.(FH)" (M.Sc. equivalent) at University of Applied Sciences Technikum Vienna, Austria
1998	Graduation as "Staatlich Geprüfter Lehrwart" (Instructor) with distinction at BAfL Graz, Austria
1990	High School Diploma (*Matura*) at BG/BRG Leoben I, Austria

Employment

since 2008	Research Assistant at the Institute of Computer Languages, Compilers and Languages Group at Vienna University of Technology, Austria

2005 - 2007	Research Assistant at the Institute of Information Systems, Distributed Systems Group at Vienna University of Technology, Austria
since 2005	Lecturer for Computer Science and Electronics at University of Applied Sciences Technikum Vienna, Austria
2000 - 2002	CEO Artecs Internet Development GmbH, Vienna, Austria
1999 - 2000	CTO Artecs Internet Development GmbH, Vienna, Austria
1995 - 1998	Software Developer, Security Auditor, 2nd Level Support at KPMG Vienna, Austria
1992 - 1995	Freelance Software Developer in several international projects

Academic Experience

11 peer-reviewed and invited papers; 10 × PC member at international conferences and events; various lecture courses at Vienna University of Technology, and University of Applied Sciences Technikum Vienna, in the field of embedded systems programming, distributed systems, physics and simulation, and artificial intelligence.

Research Interests

Distributed real-time embedded systems, model driven development, software synthesis, static analysis; Robotics, language support and modeling of robust reactive systems; Bio-mimetic and bio-inspired computing and architectures for dependable systems and cognitive robotics.

Academic Project Experience

since 2008	ALL-TIMES: Integrating European Timing Analysis Technology (EU FP7)
2005 – 2007	COMPASS: Component Based Automotive System Software (FIT-IT)