Institut für Informatik
der Technischen Universität München

DeSyRe: Decomposition of Systems and their Requirements — Transition from System to Subsystem using a Criteria Catalogue and Systematic Requirements Refinement

Birgit Penzenstadler

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

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Michael Beetz, Ph.D.

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Dr. h.c. Manfred Broy
2. Univ.-Prof. Dr. Barbara Paech, Ruprecht-Karls-Universität Heidelberg

Die Dissertation wurde am 21.10.2010 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 19.12.2010 angenommen.

Abstract

In software systems development, companies try to handle the increasing size and complexity of their systems by signing up different subcontractors for subsystems. For distributed development and smooth integration, a major challenge is to deduce *sub*system specifications from system specifications in order to deliver them to the subcontractors. Thereby, thorough requirements engineering lays the basis for successful systems' development in such a divide-and-conquer approach in order to provide a subcontractor with all information they need.

Missing information within the subsystem requirements is the pitfall for successful distributed development, so that either the subsystem requirements do not fulfill the overall system requirements completely, or there is a mismatch between subsystems during integration due to inconsistencies between the specifications for the respective subsystems.

Consequently, the research objective of this work is to investigate how a requirements engineer can systematically derive *sub*system requirements specifications from system requirements specifications. The guiding questions are:

- What is a good way for the system architect to obtain the initial system decomposition?
- What is a good way for the requirements engineer to deduce subsystem requirements from system requirements?
- How do the requirements engineer and the system architect perform both the decomposition and deduction during the requirements specification development process?

Currently, there is no encompassing approach in the literature that provides guidance to systematic decomposition of systems and refinement of their requirements to avoid loss of information.

This dissertation provides such guidance by means of a reference catalogue of decomposition criteria and an approach to requirements decomposition and refinement. The contributions are:

- A reference criteria catalogue for initial system decomposition that serves as extensive checklist during the first design step
- An approach to systematically derive subsystem requirements from system requirements by use of assumption / guarantee reasoning and decomposition patterns
- A process that exemplarily guides the application of the approach using a requirements artifact model

The results are demonstrated using a running example from the automotive domain and evaluated in an industrial case study with respect to applicability.

Contents

List of Figures

List of Tables

Acknowledgements

A big “thank you” to all the wonderful people in my life! When life is rushing, I feel that I don’t say “thank you” often enough to everybody who deserves it — this is a good opportunity.

I would like to thank many people for supporting me during the time of my thesis, either for giving me feedback on ideas, first sketches and written contents, or for simply being there for me when I had a rough time.

My advisors, Prof. Manfred Broy, for offering advice and providing feedback whenever I asked for it as well as discussions on my plans for the future, and Prof. Barbara Päch, for participating as second advisor and for a detailed and very helpful review of a draft version of the thesis.

My colleagues — about half of the comments are to be read with a big :-) behind.

- Daniel Méndez Fernández, for putting up with all my moods in the office, whether exhilarated and giggly or exhausted, for many discussions and insights on requirements engineering, for calming me down when something had upset me or made me nervous, for inadvertently teaching me how to make fun with all industrial research partners to ease the atmosphere in a stressful project, and for making me laugh a lot.
- Dr. Sabine Rittmann, for discussions about how to approach a PhD, helping me structure my ideas, for baking great cakes, for having lots of fun and being a really good friend.
- Silke Müller, for keeping an overview of everything that is going on at the research group, for managing the appointment schedule and keeping everybody happy.
- Dr. Stefan Wagner, for teaching me how to do proper research and how to approach measuring quality, and for good music.
- Dr. Leonid Kof, for giving feedback on my first draft of the thesis and being a “pain in the ass” so we got that integration of the requirements engineering approaches at our research group written down.
- Dr. Wassiou Sitou, for sharing a quite intense research project and discussion on plans for the future. My offer to teach at your future university in Togo will not be forgotten.
- Dr. Florian Deißenböck, for teaching me about software quality and that all modeling stuff finally has to be transformed into code if we want to produce software, for reviewing, and for putting up with my resistance to dress codes.
- Florian Hölzl, for making me think a lot about how to realize theoretical or manual approaches in a tool and at least partially automate them and how to ensure consistency.

- Dr. habil. Bernhard Schätz, for teaching me not to worry about the sentences “I don’t know yet.” and “I don’t understand that.”, for interesting discussions on many topics, and for the nice habit of inviting for cocktails after colleagues helped out with reviews.
- Dr. Martin Feilkas, for supporting me in the newly installed automotive lab and the related lectures and tutorials and for picking me up for BBQ when I had a flat tyre on my bike.
- Elmar Jürgens, for providing cautious advice on how I could structure my thoughts when I didn’t know how to write them down, for inspiring us with short catchy paper titles, and for a great new year’s eve party.
- Mario Gleirscher, for valuable discussions on research work and the future, and for a few great hiking trips, best one right in Stubai where we got half a meter of snow overnight in the middle of July.
- Franz Huber, Dieter Mletzko and their crew, for providing everything we need in terms of hardware and software support.
- Klaus Lochmann, for accompanying me in a number of research projects and cheering us up with really dry jokes when we at least expected them.
- Dr. David Cruz, for discussion on software architecture and methodology in general, for writing a technical report with me and thereby helping me to put my ideas into a bigger perspective for the first time.
- Christian Leuxner, for reviewing, for telling me that my first draft of the formalization was really crappy and for patiently explaining why.
- Bernd Spanfelner, for further discussions on formalization and good tea.
- Lars Heinemann, for reviewing, for being enthusiastic about our running team for the Business Run and for not being angry that I could not come to his birthday party two years in a row.
- Doris Wild, for discussions about specifications and cheering for us at Tegernsee half marathon.
- Sascha Schwind, for supporting me with the coordination of the master course in automotive software engineering and the respective lectures and tutorials.
- Benjamin Hummel, for reviewing, helpful comments, and incredible two-colored mousse au chocolat.
- Alik Harhurin, for celebrating my birthday in a strange little bar in Kyoto on the night before my first conference although he was drop dead tired from the flight.
- Jorge Fox, for practicing Spanish with me and inspiring me with his around-the-world post doc.
- Dagmar Koss, for discussions about compatibility, co-authoring my first main conference paper, and trying out slacklining for the first time with me, right between the trees behind the faculty.

- Dr. Marco Kuhrmann, for making me reflect about process integration and taking my mind off stressful thoughts with stupid jokes.

My family, my parents Sabine and Fritz Penzenstadler, for always supporting but never pushing me, and my brothers Simon and Veit Penzenstadler.

A special "thank you" goes to my friends

- Iris Aue, mi rubia, for calling me in the middle of the night (I mean it!) to share something important, for discussions on software engineering in practice, for a great vacation wave riding the French coast, for reviewing my thesis, for being there whenever I need a friend, and for girly evenings — they are *so* important.
- Christina Mohr, for mute understanding, mutual comforting and lots of laughter since we were kids — we'll still be sitting together giggling when we are grandmas!
- Michael Schölz, for founding the Munich Performa Crew[1] with me, base camp for a lot of training and even more fun together, and for deep thoughts on love and life. Furthermore his wonderful wife Evi Ruhland, and their kids Elisabeth and David, for great ("adopted") family time.
- Johannes Maifeld, for teaching me that the world and the people we interact with are mostly alright just how they are, only sometimes it takes a little more patience with them. And for great climbing and biking tours as well as sharing the perfect hammock lounge apartment with Julia and me.
- Julia Roelofsen, my princess and flatmate, thanks for an incredible trip to Costa Rica that we will never forget, many hours of shared thoughts, and fun sports time.
- Martin Glas, for discussions on technology, dissertations, software development from an engineer's point of view during many runs through the English Garden in Munich.
- Jasmin Drescher, for inspiring me, for beautiful stories (keep publishing, love!), for reminding me of the beauty of the German language, for sharing my struggle between creativity and business as well as touring the world and nesting.
- Marion and Oliver Hanke (and now, little Julius), for our shared Passau university time and for funny game evenings.
- Markus and Barbara Reschka (and now, little Peter), for being great (former) flatmates and for letting me invite myself every once in a while.
- Julius Donnert, for learning Japanese with me during two years (although I gave up afterwards), and for a great marathon in Switzerland and an awesome week of hiking on the camino de Santiago de Compostela.

[1]`http://www.munichperformacrew.de`

- Jens Dobrindt, for always setting up the most adventurous tours — 1000 meters of altitude by mountain bike, then 500 more hiking, then 1000 more climbing? Count me in!
- Janine Simons, for flowing with me on the yoga cloud, for making me feel at peace with myself and the world, for partying and sharing dreams.
- J. Fernando Zuñiga Navarrez, for my Mexican friend, for teaching me that love is unconditional — cúidate!
- Kira Montes de Oca, y su familia Pedro y pequeño Fabián, for a wonderful time in Mexico and proving that friendships last around the world.
- Veit Radkte, for sharing fun runs through all Munich parks — the only thing that could make us stop was when we nearly dropped to the ground because of laughing so hard after invented stories (so ridiculous that we called them "verbal diarrhea").
- Milo Mayr, who taught me that one can sleep anywhere at any time, whether it be during the hiking tour, or in front of the chimney while sharing a bottle of wine with a bunch of friends, but you're always up for getting together and doing something.
- Silvia and Harald Irl, and their kids Madita and Moritz, for cookie baking sessions every Christmas and for fun family time.
- Dr. Daniel König, Dr. Philipp Mai, Dr. Martin Bischoff, and Tobias Kellner, who I first met on my first summer school of the "Studienstiftung des Deutschen Volkes" in St. Johann, for great mountain tours and inspiring thoughts on academic careers and life in general.
- Ruth Eichner, who I met on my second summer school, for starting our first climbing course together, and for constantly exploring the world at every chance you get to do so.
- Carolin Heilmann, Herbert Perchtold, Bernd Müller, for sweaty karate training sessions — you were a great support for the black belt exam.
- Mattias Fuchs, for succeeding in making me lead my first vertical climbs of 30m of altitude at a time and for helping my conquer the fear with concentration and patience.
- Sabine and Steff Kratzer, together with their kids Jona and little Luis, for being friends for a long time and making me feel at home whenever I meet you.
- Dr. Karl Niederhofer, my godfather, for giving me one really good reason for doing a PhD: "When you have a doctor's degree, you can ask stupid questions any time, and nobody will give you a disapproving look."
- Thea Tsiklauri, for insights on cultural differences in business and for great Italian cuisine.
- Christine Koch, for Greek flaire, for teaching me about intercultural communication before I went to Japan and China, and for funny evenings.

- Martin Mauch, although I've already known you since I started university, now we finally got back in touch and I enjoy it a lot. Thanks for bringing me back to learning how to play the guitar.

Furthermore, there were research partners and friends who I would like to thank:

- David Callele, for a great performance at the "RE top model(ing) contest" and for inspiring talks, mostly via Skype (thanks to you being in Canada), and thanks for writing the experimentation paper with me and for sharing the PhD struggle.
- Dr. Mark Müller, Thorsten Weyer, Dr. Ernst Sikora, Jörg Leuser, and Dr. Peter Braun, my research partners during the REMsES project, for finishing this project with me successfully. And Mark, for inspiring the idea of becoming professor.
- Dr. Gerd Beneken, for teaching me on software architecture and supporting the idea of becoming a professor. We'll see...
- Dr. Stefan Kriebel, for challenging my ideas and teaching me about testing.
- Prof. Mike Mannion and Juha Savolainen, for inspiring me during the REUSE conference, where I had my first main conference paper, and for a fun party night in Beijing (including Erdinger Weißbier at the Suisse hotel beer garden).
- Prof. José Luis Barros Justo and Prof. Juan Llorens, for taking me into the Spanish community at the REUSE conference.
- Dr. Samuel Fricker, for teaching me on advisors and on personal goals during the PhD.
- Kzrysztof Wnuk, for making funny animal faces in Atlanta, for always being up for anything, and for good ideas in research — we still have to write that paper.
- Alejandro Russo Ph.D., for a great Argentinian / Swedish connection with Munich after the Marktoberdorf summer school. I will definitely come around to visit Argentina one of these days.
- Dr. Gabriele Haller, Gabriele Frenzel, and Tobias Schlosser, for discussions on soft skills and practical insights on the topic.
- Björn Saballus, for sharing the CeDoSIA (Center of Doctoral Studies in Informatics and its Applications) program with me.
- Samira Salman and Andreas Györy, for being great master students and for setting up an example for others.

Finally, a few of the teachers who have inspired me:

- Lucas Rockwood, Jeffrey Sachs, Lothar Ratschke, Fritz Oblinger, Gabriele Bozic, Antje Schäfer, Kevin Gianni, and Sara Avant Stover, for inspiring thoughts and lessons on yoga, karate, lifestyle, and the universe in general.
- Prof. Gregor Snelting [Sne98]: When I said that I didn't want to do a PhD I meant it; it took me half a year to change my mind, but you definitely helped in achieving this change.

Chapter 1

Introduction

Contents

The topic of the thesis is the transition from system requirements to subsystem requirements. This chapter explains the motivation and the problem (Sec. 1.1), describes the derived research questions (Sec. 1.2), the research design (Sec. 1.3), and the contribution (Sec. 1.4).

1.1 Motivation and Problem Statement

In systems development, which includes hardware and software, systems are growing, not only in size in terms of lines of code, but also in complexity, degree of heterogeneity, number of peripheral devices etc. To handle the increasing complexity of systems, counter measures are required.

One central counter measure is the decomposition into subsystems. In explicit, companies try to master the increasing size and costs of their systems by concentrating on their core competences and signing up different subcontractors for the development of subsystems. Thereby, the work is distributed in a divide-and-conquer approach. However, successful conquering implies a number of preconditions, namely for the initial system decomposition and for requirements engineering (RE). According to, inter alia, Kotonya and Sommerville, the basis for successful software development is laid by systematic RE [KS98], while Cheng and Atlee identified the need to propose adequate support to handle increasing systems scale in current RE research [CA09].

During systems development, first, a requirements specification is elaborated for the system, second, an initial decomposition is decided on in order to be able to develop the subsystems at different sites and, third, respective subsystem requirements specifications have to be deduced. The central role for the first step is the *requirements engineer*; for the second step it is the *system architect*; for the third step it is again the requirements engineer as the coarse-grained system design of the overall system is an input for requirements engineering for the subsystems.[1] This close connection of requirements engineering and system design implies adequate collaboration between the two developer roles to provide consistent subsystem requirements specifications.

In this work, the standard IEEE definitions of the terms *system* and *subsystem*, *specification* and *requirements specification* [JM90] are used:

DEFINITION 1.1 *System. A system is a collection of components organized to accomplish a specific function or set of functions.* □

DEFINITION 1.2 *Subsystem. A secondary or subordinate system within a larger system.* □

DEFINITION 1.3 *Specification. A document that specifies, in a complete, precise, verifiable manner, the requirements, design, behavior, or other characteristic of a system or component, and, often, the procedures for determining whether these provisions have been satisfied.* □

DEFINITION 1.4 *Requirements Specification. A document that specifies the requirements for a system or component. Typically included are functional requirements, performance requirements, interface requirements, design requirements, and development standards.* □

For distributed development and smooth integration, a major challenge is the appropriate deduction of subsystem requirements specifications from system requirements specifications in order to deliver them to the subcontractors. One pitfall for successful distributed development is missing information within the subsystem requirements. Consequently, either the subsystem requirements do not fulfill the overall system requirements completely, or there is a mismatch between subsystems during integration due to inconsistencies in the specifications.

Currently, there is no encompassing approach in the literature that provides guidance to the systematic decomposition of systems and refinement of their requirements to avoid such loss of information or inconsistency.

1.2 Research Questions

The goal of this work is to investigate the decomposition of systems and the refinement of system requirements for subsystems and to provide guidance to software development organizations and subcontractors for decomposing systems and specifying subsystems.

The research questions derived from the above problem statement were elaborated according to the recommendations given by Shaw [Sha03]. The overall research objective is:

> ***Investigate how a requirements engineer can systematically derive* sub*system requirements specifications from system requirements specifications.***

This objective is structured by means of subordinate questions:

[1]The two roles of *requirements engineer* and *system architect* are used throughout the work to denote the responsibilities for specific tasks.

RQ1 *What is a good way for the system architect to obtain the initial system decomposition?*
For system decomposition, an analysis of the potential influence criteria for system decomposition is required. This research question shows the relation of the overall research objective to architecture design.

RQ2 *What is a good way for the requirements engineer to deduce subsystem requirements from system requirements?*
For systematic deduction of subsystem requirements, the analysis investigates which cases have to be differentiated as well as which rules and patterns can be identified. This one is the most important question for the thesis at hand.

RQ3 *How do the requirements engineer and the system architect perform both the decomposition and deduction during the requirements specification development process?*
A process is required to guide the requirements engineer and system architect during system decomposition and subsystem requirements deduction on how to document and process the information contained in the specifications.

1.3 Research Design

Initially, an interview study on the state of practice was conducted to evaluate the relevance of the research questions. It was performed with participants from different companies in industry (first and second tier suppliers) either via a phone call or via email. The questionnaire examined system development, modeling, architecture, subcontractor relationships, and reuse.

To answer research question 1, the system decomposition, a preliminary draft version of a decomposition criteria catalogue was gathered from a comprehensive literature survey. This draft was included in the interview study. The practitioners approved the criteria categories and listed criteria and gave hints for further potential criteria. After the analysis of the interviews, deeper literature research and a number of discussions with fellow researchers, the catalogue was extended and a template was used to standardize the description of the criteria. Subsequently, a process for usage of the catalogue was elaborated and evaluated with case studies.

To answer research question 2, the subsystem requirements deduction, an extensive literature survey on requirements patterns and subsystem requirements derivation was performed. As none of the works provided guidance on the deduction of subsystem requirements in case of a given system decomposition, the possible cases for the decomposition structures of individual requirements were captured in patterns. The universal case was added after elaborating two special cases of the pattern that already capture the majority of occurring requirements in practice. In parallel, a subsystem model was defined as reference for the discussions on subsystem distribution and documentation. It is based on the concepts of the system model in [BSW$^+$08]. To describe how the requirements engineer actually applies the approach during system development, the decomposition and refinement were illustrated in a process that uses the requirements artifacts defined in the REMsES project [BBH$^+$09].

To answer research question 3, the application of decomposition and deduction, the concepts explained above were integrated into one process describing the tasks from a completed system requirements specification until a completed subsystem requirements specification. This guidance was then evaluated with the driver assistance case study and subsequently improved by illustrating it with a running example. Finally, the usefulness of the approach was evaluated by a second case study and its limitations were discussed.

Following the classification by Creswell [Cre03], the research design conducted for this work is a mixed approach of constructive[2] and qualitative[3] methods, where the major part is constructive.

The approach in this work was created to solve software development problems reported by practitioners, namely industrial partners from research projects. The decomposition criteria catalogue, the systematic requirements refinement and the guiding process are “innovative constructions” [Luk00] to help solve these problems.

Within this work, the parts about the study on the state of the art with the interviews (Sec. 2.2), the case study performed for DeSyRe (Sec. 6.1), and the questionnaire about the usefulness of the approach (Sec. 6.2) belong to the area of qualitative research as the small number of samples does not provide statistical relevance. Further insights were gained due to individual discussions with participants.

1.4 Contribution

The contribution of this work is to provide an approach to systematically deduce subsystem requirements specifications from system requirements specifications.

Interview Study. Seven interviews were carried out to further motivate and account for the relevance of the research objective. The results confirm that adequate system decomposition and deduction of subsystem requirements are issues of concern. Furthermore, they give insights on the state of practice in systems development, requirements engineering and management as well as system design.

Decomposition Criteria Reference Catalogue. The catalogue lists all criteria that influence system decomposition according to literature, state of the art, best practice, and experience from developers in industry. It serves as a reference when gathering and prioritizing the possibly conflicting criteria for the decomposition of a system.

[2]The constructive research approach is a research procedure for producing innovative constructions, intended to solve problems faced in the real world and, by that means, to make a contribution to the theory of the discipline in which it is applied [Luk00].

[3]Qualitative research is a field of inquiry that crosscuts disciplines and subject matters [DL05].

The catalogue contains four main categories: The *directive* criteria contain laws and standards, licensing/patents, information politics and business rules. The *functional* criteria are concerned with the usage functions the system shall provide. The *quality* criteria reflect implications by desired quality characteristics of the system. The *technical* criteria are constraints that arise from the realization platform and surrounding environment. For each criterion, the catalog lists what its impact on the decomposition is and where the related project-specific information can be found.

Approach for Subsystem Requirements Deduction. A subsystem model is defined as foundation for describing the distribution and documentation of a subsystem across abstraction levels.

On that basis, an approach is presented on how to decompose and refine system requirements according to their structure by use of assumption/guarantee specifications. This is achieved by pattern matching with two special (simpler) cases of decomposition that apply for the majority of requirements and a general decomposition pattern that applies for all requirements. Furthermore, the possibilities for decomposition and refinement of non-functional requirements are discussed.

Process for the Decomposition of Systems and Requirements (DeSyRe). This process describes the usage of the decomposition criteria catalogue and the requirements refinement when decomposing systems and developing subsystem specifications. The process is intended as a guideline for the developer and is depicted in Fig. 1.1. The system architect receives the system requirements and derives the system decomposition by use of the criteria catalogue. The requirements engineer receives the system decomposition and then derives the subsystem requirements by use of the patterns. The resulting subsystem requirements specification is realized by a subcontractor (not part of DeSyRe) and, finally, the subsystem is integrated.[4]

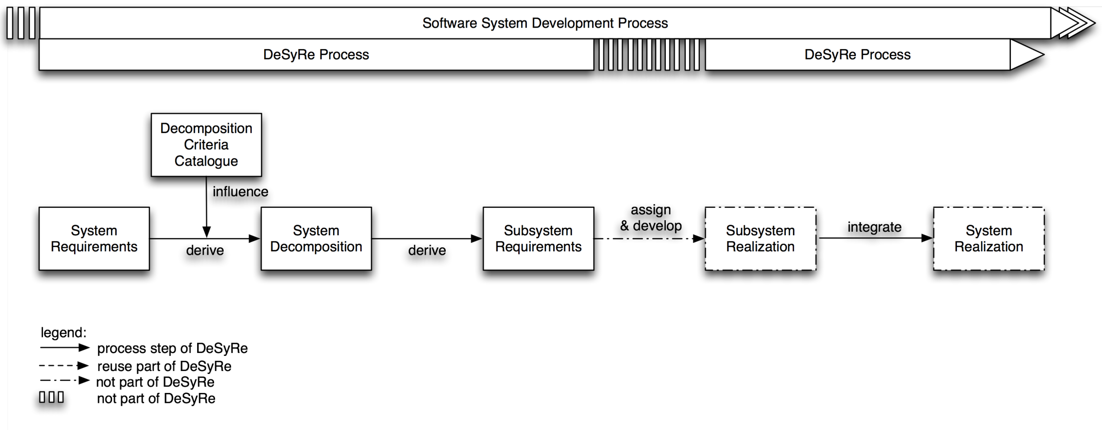


Figure 1.1: Overview of the DeSyRe method.

[4]DeSyRe includes a description of the (re-)integration in case of reuse of a subsystem requirements specification.

Case Study on Applicability. A supporting contribution is the case study on *driver assistance systems* (DAS), a real-life example from automotive industry. It serves to evaluate the applicability of the DeSyRe approach.

The DAS represent the overall system under development. From the DAS, two example subsystems are detailed, namely the *adaptive cruise control* (ACC) and the *radio frequency warning* (RFW) system. The ACC is a speed control system that automatically maintains a pre-defined speed taking into account a minimum distance to the car in front. The RFW supports the driver in coping with the information input from the surrounding environment by use of radio frequency signals.

Case Study on Usefulness. To evaluate the usefulness of the approach, a tutorial was held at a software development company and the participating developers filled out a survey form to report their appraisal.

1.5 Outline

This work is outlined as follows: Chap. 2 explains relevant background knowledge and presents the study on the state of practice to further motivate the research objective. In Chap. 3, the decomposition criteria catalogue is presented. In Chap. 4, the refinement of requirements is presented and in Chap. 5 the guiding process DeSyRe. In Chap. 6, the evaluation of the approach in the two case studies is presented. Finally, Chap. 7 concludes by giving an outlook on future work.

Previously Published Material. Parts of the contribution presented in this thesis have been published in [BGL+08], [dCP08, Chap. 3+4], [PK08] and [BFI+09, Chap. 9].

Chapter 2

State of the Art and State of the Practice

This chapter explains the general background, state of practice, concepts, and some earlier work on which this work is based on.

The following sections give some background information on software development in the automotive domain (Sec. 2.1) and present a study on the state of practice (Sec. 2.2) conducted by the author. The gained insights from that study help to better understand the problems that contractors and subcontractors currently face with respect to distributed subsystem development.

Subsequently, the chapter describes the architecture model and the requirements engineering reference model (REM). Both provided foundation for the research project REMsES (Sec. 2.5) which was a basis for this work.

Contents

2.1 State of Practice in Automotive Software Development

The work for this thesis was inspired by problems and challenges reported by project partners from original equipment manufacturers (OEMs) and first-tier subcontractors in the research project REMsES [RDSS09]. As the running example and one case study are from the automotive domain, this section gives a little background on automotive software development.

To support the driver of a vehicle in his tasks, an increasing amount of functions has to be provided, therefore the complexity of embedded systems is increasing, especially in the automotive domain [SB07]. Furthermore, the development and production of vehicles are organized in product lines, consequently the required configurability adds another dimension of complexity to software development. Additionally, strong crosslinking between the electronic control units makes designing an overall system architecture even more challenging. An appropriate architecture is one of the foundations for successful distributed development.

Crucial for the assignment of subsystems to subcontractors is efficient and precise documentation. Thereby, "efficient" means a balance between concise and easily understandable, and between capturing the important information and all possibly relevant information. The need for appropriate architectural specification and documentation is generally accepted [TA05]. However, in the automotive domain, this is complicated by strongly and widely distributed development within an association of subcontractors, where the adequate information has to be extracted from the whole system specification and distributed to the subcontractors as self-contained documents.

The current state of practice is to first produce a system specification and then again to separately produce requirements specifications for the subsystems that are to be assigned to the subcontractors. This course of action is time-consuming and costly as systematic reuse is not yet widely applied between these two process stages.

Context and Conditions. The automotive industry develops large and complex embedded systems. The original equipment manufacturers assign the development of subsystems to subcontractors. Therefore they are confronted with many challenges concerning specification, documentation, and integration until start of production (SOP).

Since the first pieces of software were introduced into cars in 1976 [PBKS07], the automotive industry has incorporated more and more software into their systems, see Fig. 2.1. The number of electronic control units (ECUs) has increased from less than 10 in 1995 to more than 60 today in some upper class cars [SB07]. Current cars feature software with up to 1 million lines of code and by 2010, premium class cars are expected to contain one gigabyte of on-board software [PBKS07].

Furthermore, the new x-by-wire technologies provide great possibilities but also great challenges for software development.

There are three major types of development in automotive engineering: research and the so-called pre-development, pre-serial development, and serial development. During research and pre-development the processes are less strict, as developers are aiming at new solutions and apply new concepts on prototype cars, which will never be released to public traffic. Pre-serial development prepares and improves the new concepts for serial maturity and serial development leads to the actual production of the cars for sale.

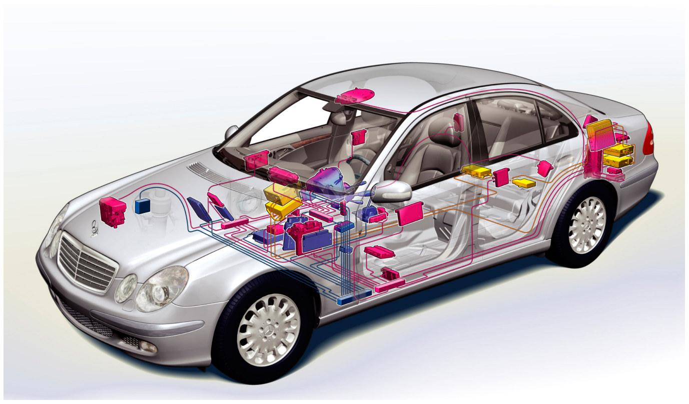


Figure 2.1: Software-supported System Components in a Contemporary Car. [BGG+06]

The prescribed process for pre-serial and serial development usually follows a standardized process model and includes the management of distributed development.

Development Process. The general automotive system (= vehicle) development process usually consists of a conceptual phase and a realization phase (see Fig. 2.2). As already mentioned in the introduction (Chap. 1), the general automotive development process is often organized according to that model. The V-Modell [Bun08b] is the obligatory development process for standard IT-projects by the German government and military service [KNR05]. The specifics of the V-Modell, however, are not of concern for this work.[1]

The V-Modell is a guideline for the planning and execution of development projects, which takes into account the whole life cycle of the system. The model defines the results that have to be prepared in a project and describes the concrete approaches that are used to achieve these results. It also defines the responsibilities of the individual participants in the project.

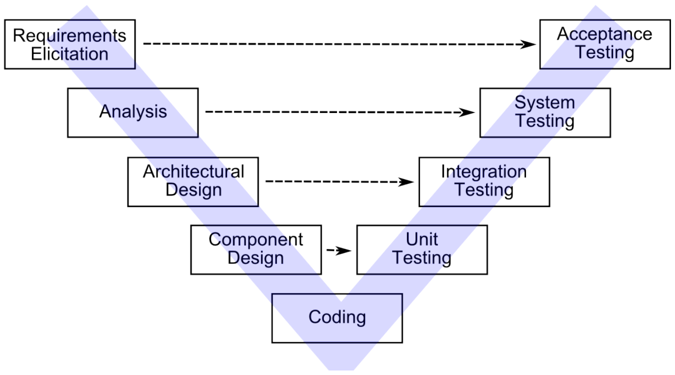


Figure 2.2: The Development Phases of the V-Modell.

[1]This work does not assume a development process exactly according to the V-Modell.

The in the context of this work most relevant part of the process model, the system development, consists of two parts: The conceptual phase, which is depicted on the left hand side of Fig. 2.2, and the realization phase, depicted on the right hand side of Fig. 2.2.

During the conceptual phase (requirements engineering & design), the requirements are elicited, and the logical architecture is designed. Then, the technical system architecture, where the building blocks are the ECUs, and the networking (i.e., the layout of the wiring harness) are defined, and the software components are specified. The components are either developed in house or assigned to subcontractors.

During the realization phase (implementation and integration), the software and/or hardware components are implemented and tested. The component test is followed by the integration, system and acceptance tests. The strict deadline is the start of production. As this development process is performed in iterations for each new car series, it will be referred to as development cycle during the rest of this work. The horizontal arrows in Fig. 2.2 indicate that verification and validation are performed at every design level.

Distributed development. This term has a double meaning and both aspects are interesting for this work. On one hand, it means distribution of labour and on the other hand distribution of software [PBKS07]. Due to the systems' size and complexity, OEMs perform highly distributed development within an association of subcontractors. The OEM decomposes the system into subsystems and assigns them to the subcontractors. Thereby, different possibilities exist to define subsystems:

1. The OEM specifies the complete system down to the technical architecture and assigns complete ECUs with the software to be deployed on that ECU to subcontractors.
2. The OEM divides the system into hardware and software and signs up different subcontractors for them.
3. The OEM decomposes the system according to functionality and assigns functional modules as performed in the aircraft domain, e.g. by Airbus[2].
4. The OEM distributes a usage function over various ECUs to save resources.

One of the difficulties lies in the fact that the subcontractor shall develop a usage function, but has to deliver a component. Another obvious challenge is the integration of all those distributedly developed components into one a-hundred-percent reliably working system.

The presently prevalent choices are versions (1) and (2), however, in the future this will presumably move to (2), (3) and (4) to allow for greater flexibility. The probably most challenging version of distributed development is deploying a function on multiple ECUs, because it requires a specified logical architecture that is completely independent from the technical architecture. This requires even better support for modeling requirements and design as the demands for appropriate documentation and communication means become more complex.

[2]`http://www.airbus.com`

Product Data Model. During systems development, automotive developers often refer to the captured requirements and design specification information as *product data model*. Product data is stored in different ways for requirements management and system design, and for each of them on different abstraction levels. According to Györy [Gyö08], it is the goal of requirements engineering to derive type series specific requirements from company goals. The abstraction levels for the requirements are brand, product line, series model, and type series.

When the requirements for a type series are documented, the top-level element is the *car system*, which is composed by *subsystems*, again on different abstraction levels. The system units that are relevant in terms of software engineering on those abstraction levels are the complete electronic system, its domain-related subsystems, the usage functions, the modules, and the components. Each of those units is composed by a number of units from the abstraction level below, e.g., a usage function is composed by modules.

This logical product data model is filled by the OEM and the subcontractors and it has to be clearly defined who fills which part of the product data model and who needs which contents.

Depending on the particular application domain or subdomain, e.g., driver assistance systems, human machine interface, engine control, the use of models and description techniques is quite different. The logical product data model is filled as concrete product data model by use of suitable representations, for example Doors, Matlab, etc. The representations for the concrete product data model have to be chosen and agreed upon by OEM and subcontractor as they need to be able to exchange the data and integrate the different representations.

In that context, AUTOSAR [AUT06] eases integration of the product data model by offering a standard for the description of a system's architecture.[3] AUTOSAR is already applied successfully by some OEMs [GGRS08].

2.2 Interview Study on the State of Practice

A small field study was performed to gain knowledge about the state of practice in the automotive domain in terms of system development, modeling, architecture, subcontractor relationships, and reuse. The major aim of the interview study was to ensure practical relevance of this work. It is structured as described and recommended by Perry [PPV00, p. 350-352].

The study was conducted in the form of a questionnaire. The questionnaire was either sent and answered by email or in an interview, depending on the availability of the respondent. The knowledge gained through the study has been used as a basis for the ideas presented in this work and for keeping in touch with the practitioners to stay aware of their actually relevant problems. The study therefore also shows the relevance of the approach presented in this work for embedded systems development.

[3]An earlier approach, EAST ADL [Lon04] is an architecture description language especially developed for automotive embedded systems but there is neither an artifact model nor has the work been continued after a first released version of the ADL in 2004.

2.2.1 Context

The questionnaire was designed with the intention to investigate the state of practice in current embedded systems development within OEM and subcontractor companies to get an insight into their processes and habits as well as challenges and problems. The participants were seven software and system developers from the OEMs BMW[4], Daimler[5], Audi[6] and MAN[7], from the subcontractors Bosch[8] and Siemens VDO[9] (later on Continental[10]) and Berghof Automationstechnik GmbH[11]. Due to confidentiality, the original answers have been made anonymous in this work.

2.2.2 Research Objective

The research objective for the interview study is defined according to the goal definition template by Wohlin, Runeson and Höst [WRH00]:

Analyze requirements engineering and management
for the purpose of validation
with respect to the state of the practice
from the point of view of the industrial developers
in the context of complex systems development.

2.2.3 Hypothesis

As the embedded systems domain is determined by large, complex systems, and most development companies are medium to big size, they are not likely to quickly adapt the latest state of the art from research. Instead, it usually takes a few years until they consider certain software engineering approaches to be sufficiently established and evaluated in order to adapt them, plus a certain amount of time to really perform that shift due to the size and organization of the development departments.

The hypotheses for the study are:

- For RE specifications, the main demand by the companies is to adhere to certain document structures.
- A rather low degree of logical modeling is performed during software development.
- The decomposition criteria (the draft version of the criteria catalogue in Chap. 3) are rated differently, but tendencies become visible.

2.2.4 Design

The questions were divided in five sections: system development, modeling, architecture, subcontractor relationships, and reuse. Within the respective

[4]`http://www.bmw.com`
[5]`http://www.daimler.com`
[6]`http://www.audi.com`
[7]`http://www.man-ag.com`
[8]`http://www.bosch.com`
[9]`http://www.vdo.de`
[10]`http://www.continental-corporation.com`
[11]`http://www.berghof-automation.de`

sections, the questions inquired the standard development process and the state of practice for methods and techniques:

System development:

- What is your general approach for software development?
- Which process do you follow?
- Which artifacts are produced during requirements engineering and design?
- How are the artifacts related to each other? For example, are specific system models derived from artifacts?
- Which tools do you use? Is there a continuous tool chain that supports the development process? Do you have tools or a tool chain that is prescribed by your company or does every department or project team handle tooling separately?
- Do you model a *logical architecture* within your current development process?[12]

Modeling:

- Which models or notations are generally used during the development process?
- Is requirements engineering performed only text-based or do you use models for support?
- Which methods and diagram types are used for documentation?
- Do you use any domain-specific methods?
- Are there company-specific guidelines that have to be obeyed during development or do you use external standards?
- Are there defined and assigned roles during development? If yes, which roles exist? Are there certain guidelines defined for these roles? Does this lead to problems?
- Is there a standard company terminology that everybody adheres to?

Architecture:

- Who is responsible for the software architecture of a system during development?
- Is there a central architecture team that plans the overall system architecture beforehand?
- Which criteria are used to decide on a system's architecture?
- Do you use templates or guidelines that support the definition of the architecture?
- How do you identify logical and/or technical subsystems?

[12]The explanation of the term *logical architecture* for the participants was: The *logical architecture* is a model of the complete functionality of a system in form of logical, communicating components while still completely abstracting from implementation decisions.

- How would you rate the importance of the criteria given in Tab. 2.1 for system decomposition?
- Are you missing any criteria in the list?

Subcontractor relationships:

- Which parts of the software is developed in-house and how much do you assign externally?
- Who coordinates the assignment of software to subcontractors?
- What are the decision criteria for the assignment of the development of subsystems to subcontractors?
- Would you assign the development of cross-cutting functionality to subcontractors?
- Does the current organizational structure of subcontractor relations influence the system's architecture? Or are subcontractors chosen according to the system's architecture?
- Which parts of the overall system specification does the subcontractor get as requirements specification?
- Does the subcontractor get black box specifications with defined interfaces or do they get to know the internal realization? In case of black box, does this often lead to call backs or does this work well?
- How do you document feature interaction with subsystems assigned to other subcontractors?
- Are there currently communication problems or other organizational challenges with subcontractors?

Reuse:

- Where do you currently perform reuse? To what extent? Which artifacts are reused?
- Do you have internal guidelines for reuse?
- Do you perform reuse in cooperation with your subcontractors or separately?

As the participants were concerned about not giving away sensitive information, their answers and the results from the interviews are presented in an anonymized fashion.

The data was collected in emails and during interviews, face-to-face and per telephone, between October 2007 and October 2008. Seven participants answered the questionnaire. The following section presents the major results.

2.2.5 Results

Before interpreting or analyzing the results, the following bullet list summarizes the most important statements and answers received from the participants:

- There is a defined software development process in every company.

- RE specifications are mainly text-based, sometimes UML diagrams are used.
- The tooling is diverse, with products from, inter alia, Microsoft, Telelogic, and Vector, as well as in-house developed tools.
- The rationale, for example for the decomposition of the system, is usually not documented at all.
- A logical modeling of the system is often skipped for early modeling of the technical architecture.
- Influences from the OEM-subcontractor relationships exist in both directions and efficient communication of requirements and constraints is a challenge.

The other result of high interest is the prioritization of the different criteria, that have to be considered when decomposing a system. The weights that the interviewees assigned to the criteria are summarized in Tab. 2.1. Thereby, the listing of criteria was given by the questionnaire and the weights to be assigned were between one and three. The respective part of the questionnaire was filled out by 5 participants.

Table 2.1: Table of Decomposition Criteria and Assigned Weights.

Functional criteria	Logical clustering according to usage	8
	Dependencies	11
	Interaction	10
Architectural criteria	Communication requirements	15
	Technical constraints	12
	Design rules	9
Directive criteria	Laws and standards	10
	Patents, licenses, certificates	8
	Business rules, information politics	4
	Implications from subcontractor relationships	10
Quality criteria	Performance	14
	Correctness, robustness, reliability	14
	Usability	8
	Maintainability	12
	Security	12
	Costs	15

2.2.6 Analysis

All three hypotheses were confirmed within the answers. First, for RE specifications, the main demand by the companies is to adhere to certain document structures. Requirements are documented using natural language, and sometimes UML, but no rationale is captured explicitly. However, the participants expressed interest in more guidance on that topic.

Second, the hypothesis about a low degree of logical modeling that was being performed proved to be right. Instead, the companies start straight away with technical modeling of their systems, thereby strongly restricting the solution space.

Third, the decomposition criteria catalogue was perceived as complete, and some participants provided suggestions for further investigation and encouragement to analyze and describe the criteria in detail. There is an emphasis on quality and especially on costs. What the listing did not represent were the dependencies between some of the criteria.

2.2.7 Validity of the Study

The internal validity of the study is given as the results are direct citations from the answers and the table with the assigned weights for the decomposition criteria was simply summed up for all participants. The goal of the study was to get a feeling or general understanding for the practitioners' views on the questions mentioned above and their state of practice in system development, modeling, architecture, subcontractor relationships, and reuse.

The major threat to external validity for the study at hand is that with such a small number of participants it can not be considered representative. The author is aware that a study of that size does not have any statistical relevance. Therefore, the results in Sec. 2.2.5 were not given in statements with percentages as this would imply assuming statistical relevance.

However, for the purpose of getting an insight into the state of practice the study was adequately sized as there were interviewees from seven different companies, both OEMs and subcontractors.

2.2.8 Conclusions

The answers given in the questionnaire and the interviews serve as confirmation for the practical relevance of this work.

As the decomposition criteria catalogue was perceived as complete, it can be used as a basis to further investigate on, refine and enhance the decomposition. The resulting catalogue and the interrelation of the criteria are presented in Chap. 3.

Proposed modeling approaches will only be successful if they are adaptable to different tools. Big industrial companies are in general very reluctant to change their tooling (whether commercial or developed in-house). Therefore, this work does not prescribe the use of a specific tool, but it can be adapted for a variety of tools already in use.

2.3 Software Systems Architecture Model

The architecture model by Broy et al. [BSW$^+$08] represents the basic understanding of model-based development for this work. The concept of abstraction levels allows for different views onto the same system with emphasis on certain aspects that have to be considered during development. At the same time, they still abstract from other aspects that have to be dealt with at lower levels of abstraction. This separation of concerns makes it easier to concentrate on one aspect at a time.

A general overview of the model is provided in Fig. 2.3. Three abstraction levels are in use: The *usage level* shows the results of requirements engineering including the functional hierarchy with usage interfaces (black-box view), the *logical architecture level* models represent the structure and internal behavior of the system (white-box view), and the *technical architecture level* adds

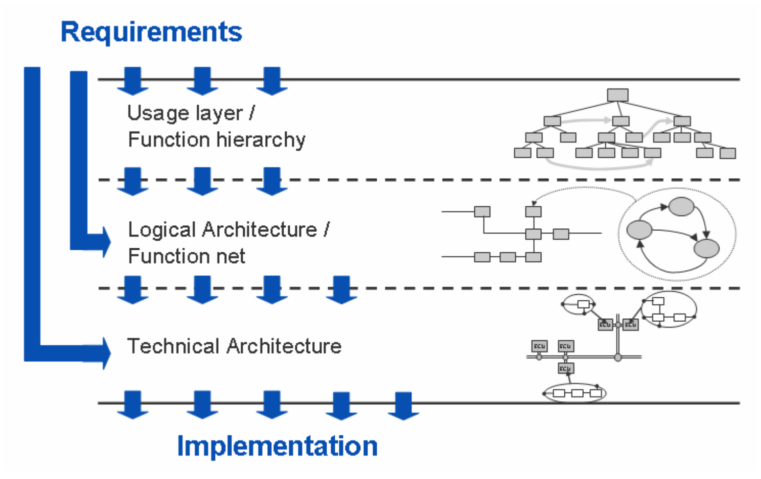

Requirements
Usage layer / Function hierarchy
Logical Architecture / Function net
Technical Architecture
Implementation

Figure 2.3: Abstraction Levels of the System, Illustration from [BSW+08].

code and task models for the software realization and deals with issues concerning the hardware realization [BSW+08]. Over the abstraction levels, the focus shifts from the complete system to function groups to realization components, providing more detail on each level and moving from abstract and implementation-independent to concrete and technical. On each level, the model is enriched by additional information. Similar abstraction levels are used by Große-Rhode et al. [GRM04].

The partitioning of the levels is conducted in that form to meet the specific challenges of developing embedded system. Each level uses suitable models for describing the aspects of those challenges. The abstraction levels focus on different types of (horizontal) relations and interactions between the respective entities. Furthermore, the vertical relationships between the levels can be followed through the use of a tracing model.

The Usage Level. The usage level offers models that allow to formalize functional requirements, represent them as hierarchical relations and additionally show dependencies between those functional requirements. The models on this level reduce the system's complexity and structure the usage behavior. This is the basis for detecting interaction early during development.

On the usage level, the system borders of the system to be specified are defined. This includes the interface definition to the external environment (driver, road, etc.) and the identification and definition of interfaces to surrounding systems that interact with the system to be specified. The behavior of the whole system is specified in a black box view, i.e. the message exchange at the system border to the environment is determined. Thereby, the behavior of the system is captured in a structured way as *service model*, which is composed by a hierarchy of services and their interrelations.[13]

Definition 2.1 *A service is a partial piece of behavior which can be observed at the system boundary. [Rit08b].* □

[13]The service model is defined in detail by Rittmann [Rit08b].

The Logical Architecture. The logical architecture level offers models that allow to structure the functionality in logical components. The formalized functional requirements from the usage level are realized through a *network of hierarchical components*, which are still independent of the underlying hardware. The model of the system on the logical architecture level is executable and can be simulated. Therefore, the logical architecture is available for early validation. The modularization and hardware independence of this level reduce the complexity of the model and create a high reuse potential.

Compared to the usage level, the logical architecture does not concentrate on the formalization of externally visible functionality but on structuring the system in terms of logical, communicating units, whose behavior in total realizes the one defined on the usage level. The structuring into logical components is, in general, independent from the hierarchical decomposition of the usage functionality. The components are modular units that per se can be developed separately and then reassembled as the desired system according to the model of the logical architecture. A logical component is involved in providing a service of the usage level. In general, there is an n:m-relationship between services and logical components.

The Technical Architecture. The technical architecture abstractly describes the realization, which is composed by hardware and software. It offers suitable models to describe the behavior of hardware and software and allow for description of the used hardware's influence on the system behavior. Thereby, the level of abstraction is chosen such that statements about the compliance to realtime requirements are possible and no further changes of behavior occur during the subsequent transition from technical architecture to implementation, but only software-related transformations (e.g. middleware procedure calls).

The models of the technical architecture level realize the logical architecture's specified behavior in a system of software and hardware. For this purpose, the model is split into two parts: One part contains the partitions of the model that shall be realized in hardware, the other one contains the application-specific parts that shall be realized in software. The application-specific part is divided into partitions (= cluster) that are executed on the respective component of the platform. The term *platform* denominates hardware and software that is used for application integration (e.g., drivers, middleware, ...).

The separation of hardware independence and hardware dependency in the whole model leads to a high reuse potential on every abstraction level. This architecture model was also used as a basis for developing the artifact model of the REMsES project [BGG$^+$07] that is presented in Sec. 2.5.

2.4 Requirements Engineering Reference Model

The Requirements Engineering Reference Model (REM) by Geisberger et al. [GBB$^+$06] is a requirements engineering framework with an artifact model at its centre. It was developed in cooperation with Siemens Research at Princeton. The artifacts (see Fig. 2.4) are the work results of the RE activities in product or product-line development and have refinement relations and

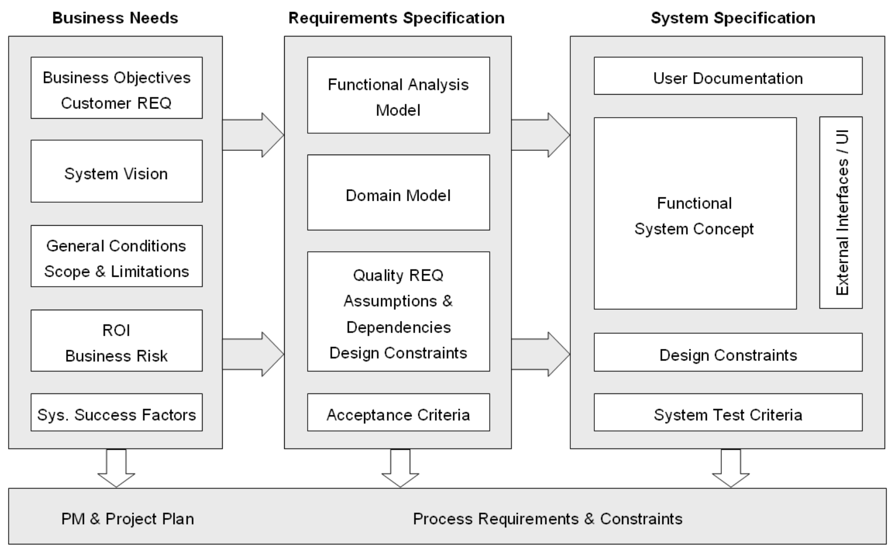

Business Needs
Requirements Specification
System Specification
Business Objectives Customer REQ
System Vision
General Conditions Scope & Limitations
ROI Business Risk
Sys. Success Factors
Functional Analysis Model
Domain Model
Quality REQ Assumptions & Dependencies Design Constraints
Acceptance Criteria
User Documentation
Functional System Concept
External Interfaces / UI
Design Constraints
System Test Criteria
PM & Project Plan
Process Requirements & Constraints

Figure 2.4: REM Artifact Model [GBB+06].

dependencies between them. The framework is completed by a role model for the responsibilities and a tailoring approach for the set-up.

The integration of the development of business needs, requirements specification and system specification provides a comprehensive approach to goal- and system-oriented requirements engineering that includes all participating stakeholders and their different views on the system. Examples for those stakeholders are marketing people, managers, system engineers, mechanics, and future users. Their perspectives on the system result in different kinds of requirements which are all integrated in the overall artifact model. REM reflects the influences of different stakeholders on the system and includes all information that reflects their respective needs and goals. REM was developed in the domain of embedded systems but is also applicable to other realms of systems and software engineering. The artifact model (see Fig. 2.4) is structured in three parts: Business Needs, Requirements Specification, and System Specification.

Business Needs. The Business Needs specify customer and strategic requirements, including product and business goals of the system development. It consists of the following artifacts:

- Business Objectives and Customer Requirements: product market positioning and customer requirements.
- System Vision: a list of main features and assumptions/dependencies of the planned product or product line.
- General Conditions and Scope & Limitations: high-level non-functional requirements and the delimited scope of the application domain or product line.
- ROI and Business Risk: cost/benefit, expected sales revenue, development and launch costs, and risk analysis

- System Success Factors: how the system is judged to be successful.

Requirements Specification. The Requirements Specification contains the product functional and non-functional requirements. They are analyzed and modeled from the customer and user perspective and derived from (and justified by) the Business Needs. It comprises the following artifacts:

- Functional Analysis Models: analysis and description models of the business and application processes and scenarios.
- Domain Model: structured specification of the application domain and its characteristics.
- Non-functional Requirements Model: quality requirements, assumptions, dependencies, and design constraints.
- Acceptance Criteria: specification of the acceptance criteria for testing the system.

System Specification. The System Specification contains a detailed definition of the functional system concept, the required behavior of the considered system and its integration into the overall system and environment. It defines constraints to the detailed design and realization of the system (software, hardware - electrical, mechanical). The artifacts include:

- User Interface Specification / User Documentation: description of how the user will use the system.
- Functional System Concept: detailed functional system requirements for services, interaction, behavior, data and usage constraints.
- External Interface Specification: interface specification of interacting systems / components of the domain or used software and hardware components.
- Design Constraints: limitations to the further detailed design and realization of the specified system concept.
- System Test Criteria: acceptance criteria and test cases for system integration and validation.

The document structure for a specific product development project is usually tailored to meet the organization's process definition. The three blocks of REM are independent from the abstraction levels presented in Sec. 2.3, in contrast, they differentiate the documented contents. The REM model served as one of the inspiration sources for the REMsES artifact model (see Sec. 2.5) used in this work.

2.5 The REMsES Project

The REMsES project [BBH+09] was a research collaboration with partners from academia and industry. The goal of the project was the elaboration of a validated, practical guide for systematic requirements engineering and

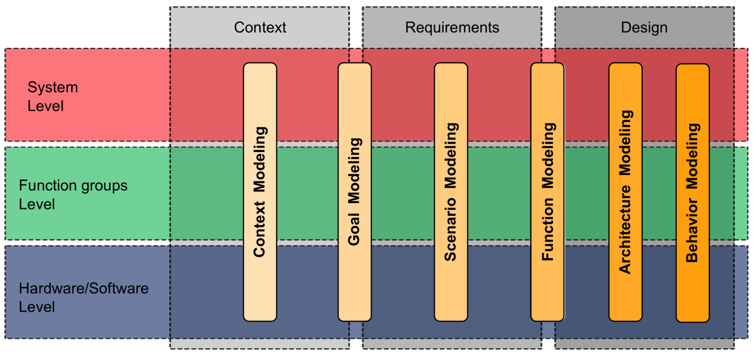


Figure 2.5: Structure of the REMsES Artifact Reference Model with Assigned Specification Techniques.

management of embedded systems, especially in the automotive domain, on the basis of a differentiated product model.

The participating partners were the Technische Universität München, the Universität Duisburg-Essen, the Daimler AG, the Robert Bosch GmbH, the Berghof Automationstechnik GmbH, and the Validas AG.

The result is an artifact-based approach with a reference artifact model. The model combines some of the ideas of the architecture model's abstraction levels introduced in Sec. 2.3 and the content categories introduced in Sec. 2.4.

2.5.1 Structure Concepts

The reference model is based on two key concepts: support for abstraction levels and coverage of three content categories. The structure supports requirements engineers in determining which type of model they should use and what kind of abstractions they should create in a particular project situation. The structure is made up of two conceptual dimensions that are based on the two key concepts (see Fig. 2.5), resulting in a 3*3-matrix structure with nine artifact classes.

Support for Abstraction Levels

Requirements at different levels of detail and abstraction (see Fig. 2.5), ranging from business goals (within the system level context) to detailed technical requirements (within the SW/HW level requirements), need to be included in the requirements document of an embedded system. High-level requirements provide a justification of detailed requirements and support the understandability of the requirements document. Low-level (detailed) requirements are needed to provide enough information for implementing the system correctly.[14]

[14]Within model-driven development of embedded systems, the Rich Components approach by Damm [DVM+05] focusses on improving reuse in the embedded domain. The components are thereby modeled on the abstraction level of technical architecture with explicit assumptions including confidence levels, but the approach includes neither explicit documentation of requirements nor higher levels of abstraction. Metropolis (see `http://www.gigascale.org/metropolis/`) is a formal design environment for heterogeneous systems, where the starting point is formal function modeling, therefore no support is provided for requirements

However, this diversity of requirements at different levels of detail demands a systematic way for dealing with each requirement adequately according to its level of detail. The granularity of a requirement influences, for instance, its importance in a certain stage of system development. Abstraction levels allow for different views onto the same system with emphasis on certain aspects that have to be considered during development, while still abstracting from other aspects that have to be dealt with at lower levels of abstraction.

In the REMsES project, a hierarchy of three abstraction levels was adopted: *System Level*, *Function Groups Level*, and *Hardware/Software Level*, depicted as horizontal rectangles in Fig. 2.5. These levels are the result of an extensive survey on the abstraction levels defined in existing approaches and used in practice. The abstraction levels form the vertical dimension of the structure shown in Fig. 2.5. The three abstraction levels used in this work (Fig. 2.6) are based on [BSW^{+}08] (see Sec. 2.3): The **usage level** shows the results

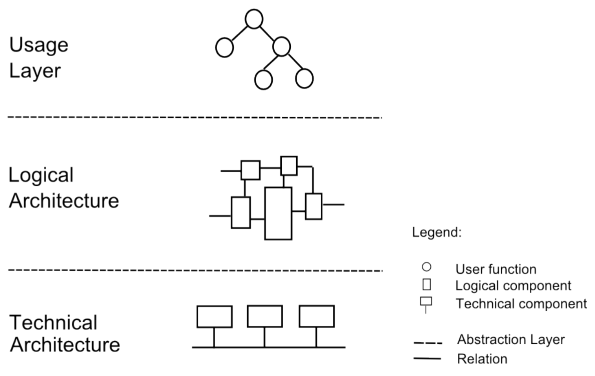


Figure 2.6: Abstraction Levels of the System.

of requirements engineering including the functional hierarchy with usage interfaces (black-box view), the models of the **logical architecture level** represent the structure and internal behavior of the system (white-box view), and the **technical architecture level** adds the code and task models and deals with issues concerning the hardware realization. These levels were generally introduced in Sec. 2.3, and are described in their application within the REMsES approach in the following.

System Level. At this level, the stakeholders take an *outside*, i.e., a black box view of the system. Requirements artifacts modeled at this level focus on the usage of the system through its human users and other systems. The artifacts capture usage goals, usage scenarios, and the functions or services that the system offers to its users via defined interfaces. These services represent the functionality that is directly visible to the users, i.e. it should not comprise system-internal or auxiliary functions.

engineering. The services are then bundled in interfaces, decomposed into sequences of events, and mapped onto a network of media which corresponds to the implementation platform [SV03]. Both of those works model on a technical architecture level and do not take into account the requirements engineering part of the process.

Function Groups Level. The function groups level represents a white-box view on the system. At this level, the system is viewed as a network of interacting, logical units obtained by a functional decomposition of the system. These units are referred to as "function groups". The requirements that are defined at the system level can be refined at the function groups level and assigned to individual function groups. Function groups have defined interfaces and can interact with each other as well as with the system environment. Each function group exhibits a defined behavior at its interfaces. Function groups are identified, for instance, through hierarchically decomposing and clustering the system functions that are defined at the system level. When creating models at the function groups level, requirements engineers are advised not to perform a partitioning of the system functionality into hardware and software. Thereby, the system requirements can be detailed without making premature design choices about the technical realization of the requirements. Thus, technical details pertaining to the realization of the functionality in hardware or software are disregarded at this level.

Hardware/Software Level. At this level, a coarse-grained partitioning of the system functionality into hardware and software is defined. For this purpose, the system is decomposed into (coarse-grained) hardware and software components. This decomposition can be regarded as a preliminary or draft system architecture. Thus, a high-level software architecture and a hardware topology (a high-level HW architecture) are modeled at this level. In order to reduce the complexity of the models at the HW/SW level, the considered high-level SW architecture is limited to application software components. Concerning hardware, the models should focus on peripheral devices such as sensors and actuators, i.e. those HW components that are needed to realize the interactions of the system with its environment. It should be noted that detailed design models are not in the scope of the HW/SW level. The main purpose of the architectural models is to support the detailing of the requirements. The requirements at the HW/SW level are obtained, for instance, by refining the requirements defined at the function groups level and assigning the resulting requirements to individual hardware or software components.

Coverage of three Content Categories

It is common knowledge that it is important to include the views of different stakeholders for successful requirements engineering [EYA$^+$05]. These views are not only different views onto the same system idea, but also views with a different scope. For example, a marketing person, in contrast to a system architect, has a different view on the system (i.e. the system's functionality as perceived by the user as opposed to a structural and behavioral view of the logical systems units) *and* a different scope (economics and business goals as opposed to design ideas).

By analyzing requirements documents in the automotive domain, three main content categories of such a document were identified to include these different views and scopes: context, requirements, and (high-level) design, depicted as vertical rectangles in Fig. 2.5. Therein, the categories context and design contain important information for the requirements engineering process and therefore have a considerable influence on the requirements.

The content categories relate to the horizontal dimension of the structure shown in Fig. 2.5. The three categories are defined orthogonally to the abstraction levels. In other words, each content category can be considered at each abstraction level.

Context. The context of the system is the part of the real world that influences the requirements for the system and therefore the system itself. Context artifacts are, for example, laws, business goals, general constraints, environmental conditions, information about adjacent systems etc. Many requirements for the system respectively its components directly originate from the demands and constraints imposed by the context. In other words, the context sets the frame within which the system is developed.

Requirements. Requirements can be expressed using conceptual models such as goal models (see e.g. [vL08]), scenario models (see e.g. [Coc00]) as well as models of function, data, and behavior (see e.g. [Dav93]). The industrial project partners identified goal models, scenario models, and function models as the most important models needed to support the requirements engineering process of an embedded system. Hence, these three types of models were included in the requirements artifact model for documenting system and component requirements.

Design. In the development of embedded systems, requirements and design are tightly intertwined. In other words, a certain amount of design information in the requirements document is inevitable in the embedded systems domain, because the knowledge about major system components is required to specify detailed requirements (see [PS07b]). Detailed requirements such as component requirements must be specified, for instance, to facilitate the integration of different systems developed by different subcontractors. In addition, in some cases, the contractor includes design artifacts in the requirements document to hint at the intended solution, respectively, to outline a feasible solution.

Thus, the included design artifacts in the reference model capture the major parts of the system, the essential relationships among these parts, and the behavior of these parts at their interfaces. By introducing design explicitly as a content category in the reference model, developers are encouraged to document requirements and design as separate models, rather than intermingling the two. Traceability over the content categories means the relation of a context artifact element to the respective requirements artifact element to the correspondent design artifact element. This is called horizontal traceability.

2.5.2 Specification Techniques

The structure presented in Sec. 2.5.1 defines nine coarse-grained categories of information to be included in a requirements document (three basic content categories at three abstraction levels, respectively). In addition, the artifact reference model suggests six specification techniques that are applied across all three abstraction levels (depicted as small vertical rectangles with round corners in Fig. 2.5) that requirements engineers of an embedded system can use for representing the different types of information. These specification techniques have been chosen such that they cover all three content categories (Context, Requirements, and Design) as well as all three abstraction levels (System, Function Groups, and Hardware/Software). Furthermore, the aim was to support seamless transitions between the abstraction levels and content categories as far as possible. The selection of the specification techniques can thus be seen as a first step to support an integrated model-based requirements engineering process for software-intensive embedded systems. The guide refers to specification techniques such as scenario modeling rather than to particular modeling languages (such as the UML Sequence Diagram). This allows the users of the artifact model to select a modeling language that corresponds to a chosen specification technique and, additionally, fits in the organization or project (e.g. regarding the available tool support and the experience of the developers).

In the reference artifact model, each specification technique is described using a template that comprises, amongst others, the following items: short description, advantages, example model.

Context modeling develops a model of the context or surrounding environment of a planned system, function group, or software component. The actual content of a context model can vary significantly depending on the type of context that needs to be modeled for the planned system. The model distinguishes between business context, stakeholder context, and operational context (see [WP08]).

Goal modeling provides an overview of the characteristic functional and quality properties of the system, a function group, or a software component. Goals are motivated by marketing and justify detailed requirements and design decisions but are defined independently of a specific technical solution. High-level goals can be refined hierarchically into sub goals (see, e.g., [vL01]).

Scenario modeling develops sequences of interactions that illustrate the satisfaction of goals. The technique provides support for integrated modeling of scenarios on all three abstraction levels (see [PSP09]).

Function modeling documents the functions of the planned system and their relationships. The detailed definition of a function encompasses the data on which the function operates, the events that the function reacts to or triggers, and the valid pre- and post-conditions for the execution of the function. A function model consists of an overview diagram of the functions and their relationships and template-based definitions of the individual functions (see [Grü08]).

Architecture modeling defines the essential structures of the planned system in terms of components, connectors, and interfaces, first in a logical model of the system under development, then in the technical architecture (see, e.g., [MT00]).

Behavior modeling represents a behavioral view of the system operations. This allows for a comprehensive specification of the desired system behavior. Each component is, for example, described by a state-transition diagram which models the behavior as receiving inputs and producing outputs. The latter, in general, serve as input for the next component (see, e.g., [Dav93]).

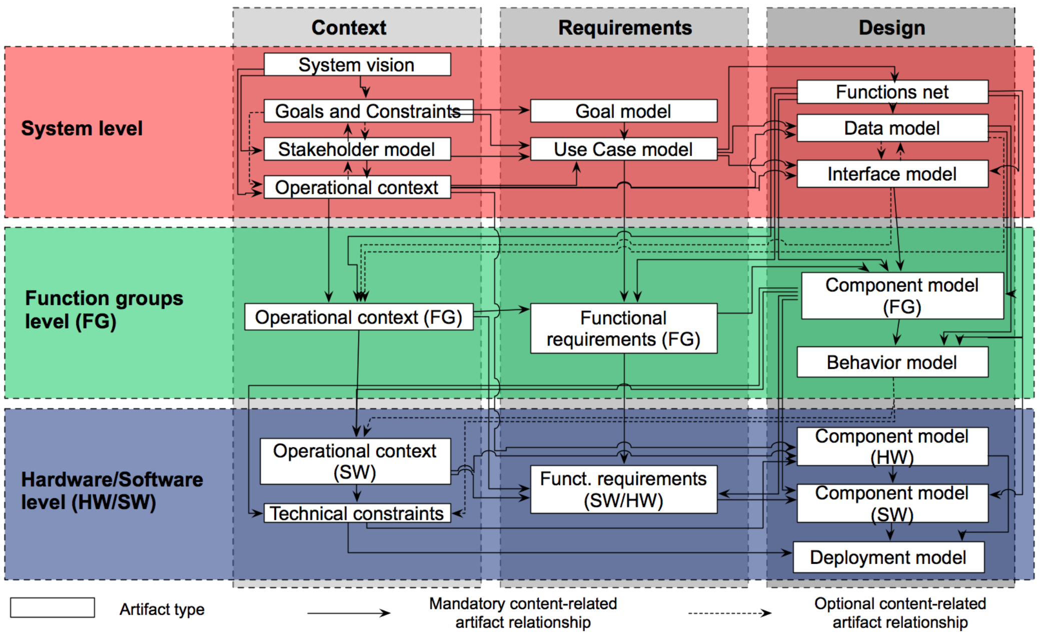


Figure 2.7: The REMsES Model's Artifacts.

2.5.3 Artifact Model

The artifact model presented was developed to serve as an easily applicable reference model in industrial software development. This section details on the model with its individual artifacts used for documenting requirements and system specifications, and furthermore discusses artifact responsibility, a mapping of the artifacts onto a document structure, and tooling. The artifacts shown in Fig. 2.7 are the basis for a systematic development of subsystem specifications.

As the method does not strictly require a specific modeling technique for the individual artifacts, different notation techniques are proposed for each of them. Independent from the chosen notation, the guidelines proposed by the OMG Reusable Asset Specification [OMG04] should be followed to provide the basis for later reuse. Examples for the artifacts are provided in the case study in Chap. 6.

Usage Level / System Level

On the Usage Level or System Level (top level in Fig. 2.7), the artifacts for the content category Context (left column in Fig. 2.7) are the System Vision, the Goals and Constraints, the Stakeholder Model, and the Operational Context. The Requirements artifacts (middle column in Fig. 2.7) are the Goal Model and the Use Cases, and the Design artifacts (right column in Fig. 2.7) are the Functions Net, the Data Model, and the Interface Model.

System Vision. The system vision is an image of the problem and aim of the system to be developed which is agreed upon by all stakeholders. It describes strikingly the alterations of reality that are induced by the system. The system vision is a textual abstract, optionally illustrated by a picture of the future system, that describes the system's core functionality and its main purpose on a high level of abstraction. It can be realized as plain text or template-based.

Goals and Constraints. The goals capture business goals and quality goals and constraints. The business goals sum up the economic and customer-related aims of the system and thereby describe, what shall be achieved through the system with respect to market and customers. Business goals originate from the decisions from customers, marketing, and management. Each business goal can be captured in simple text, so the collectivity of the business goals is a list of text blocks.

The quality goals determine specific characteristics that the system shall demonstrate apart from the actual functionality. This includes characteristics with respect to maintainability, performance, availability, usability, and security of the system. The analysis and refinement of quality goals can lead to further quality goals but also operational goals. Quality goals are derived from business goals and stakeholder demands. They are captured either as plain text or as goal graphs or goal models in the form of AND/OR trees. For a detailed description of how to develop such a goal model, see e.g. [vL01].

Stakeholder Model. The ones who typically possess knowledge about important context aspects and are crucial sources for requirements for the planned system are the stakeholders. Typical examples of stakeholders are ordering customers, system users, managers, jurists, marketing experts, engineers, system architects, system testers, and production supervisors. It is quintessential to list all relevant stakeholders involved in the requirements engineering process. Each stakeholder is described by the name of the person, organization or other group, and their respective role, and is characterized by a description which relates to his specific interest in the system under development. Their wishes and requirements can either be documented separately with the stakeholder description, or the respective artifacts capturing them can be referenced from the characterization to avoid overhead. This leads to a list of template-based tables or text blocks with references to other artifacts.

Operational Context. The operational context documents the surrounding technical and physical context of the system and their operational constraints. The technical context documents the technical embedment of the system into its environment. Part of the technical context are adjacent systems and their relations. The physical context encompasses the physical dimensions of the environment that are controlled or measured by the system, for example limit values or events. The operational environment can be captured in UML Object Diagrams [Obj07] or in a box-and-line diagram. These artifacts build the basis for developing the concrete requirements specification for the system.

Goal Model. The goal model gives a hierarchical overview over the demanded functional and quality characteristics of the planned system. The goals are derived from the system vision and issued by different stakeholders. The goal model can for example be realized using the KAOS notation.

Use Cases. Use cases describe the use of a planned system from a system user's point of view where the system user can be either a human user or another system. The artifact is usually a use case diagram that characterize the usage in detail. The description is realized by means of templates. A use case thereby substantiates one or several goals, as it documents examples for usage operations that lead to the fulfillment of goals.

A use case groups scenarios: a main scenario and a number of alternative, error, or exception scenarios. A scenario describes a linear sequence of interaction of one or more users with the system. In addition to the textual documentation in a template, the scenarios can also be formalized by Message Sequence Charts [ITU96] or behavior models.

Functions Net. The functional view is represented as functions net or *service graph* that denotes the system's functional features in a black box manner, where a service is a formalized use case. The services of the system are depicted as a net of hierarchically decomposed usage functions and their relations. A service thereby shows a behavior that is perceivable for the system user. In contrast to a model of the function, the function net depicts an overview of the interrelations of the services.

Dependencies between the services are represented through relations as for example *trigger*, *cancel*, and *subfunction* [Grü08]. The division into subfunctions does not constitute a system decomposition in interacting components but merely a hierarchical structuring of the black box functionality. The service graph is depicted in a net-like shape with nodes that represent the services and lateral arrows that depict their relations.

Interface Model. The data view is captured in an interface specification for the system interacting with the operational environment. The system interface contains all syntactical information about functions and their data types as well as references to the semantics of the functions. For every externally visible function, the input and output parameters are described including their types. Thereby the emphasis lies on "externally visible", as internal functions are not described until the logical architecture abstraction level.

Data Model. The data view analyses the data objects and data structures from the requirements. The analysis of users, system, and participating components leads to a description of the data interfaces of the data objects and at the system interface. The data view is captured in form of a data dictionary, either in a data model, for example an entity-relationship diagram, or by means of templates.

Function Groups Level / Logical Architecture Level

On the Function Groups Level or Logical Architecture Level (middle level in Fig. 2.7), the only artifact for the content category Context (left column in Fig. 2.7) is the Operational Context (FG). The Requirements artifacts (middle column in Fig. 2.7) are the Functional Requirements, and the Design artifacts (right column in Fig. 2.7) are the Component Model (FG) and the Behavior Model.

Operational Context (FG). The operational context for function groups documents the operational environment of the system functions and components to be developed. This refinement of the operational context on the complete system layer also includes logical sensors and actors.

Functional Requirements. The functional requirements can encompass function requirements, data requirements, and behavioral requirements. It depends on the type of system which description is considered the most adequate. In the case of embedded systems, function requirements are most likely used. As the use cases and scenarios describe only examples for the system usage, it is important to describe the complete functionality of the system within the requirements. At this stage, the point of view slowly traverses from the user's point of view to the system's point of view. The requirements can either be documented as structured text blocks or model-based in activity diagrams.

Component Model. The artifact encompasses a structural view on the components and their interrelation. Thereby, the collectivity of the logical components realizes the services described on the usage level. The component model gives an overview of the logical system's architecture and specifies the structure of the architecture and the logical components. In case the specification technique Focus [BS01] is used, the syntactical interface is defined by typed ports. Components are connected to each other through these ports and communication takes place via channels between these ports. Consequently, the types of the ports show up again in the interface and data specification.

The structural view is composed by the components and the connecting channels. It can be represented either in form of a hierarchical system structure diagram, a UML Component Diagram [Obj07], or as a simple box-and-line diagram with additional explanatory comments.

Behavior Model. The behavior view depicts a state-oriented examination of the usage processes (mapping of input data to output data), especially of the considered states and transitions of the system. Based on the relevant operation modes and the analysis of the usage processes, a state-oriented structuring and modeling of the usage processes is carried out. This modeling view supports, especially with regard to security issues, the identification of possible, undefined and uncontrolled system states and the respective refinement and completion of the behavior specification.

The behavior view includes a state transition automaton for each logical component of the artifact Component Model that describes its behavior. Again, if Focus [BS01] is used, the communication between components is represented in channels and ports as specified for the Component Model. Inputs to a component are received via ports; their resolution can alter the state of a component. Outputs can be generated and transferred on to other channels, and subsequently they might serve as (altered) input for other components.

A state chart consists of control states and transitions between states. A transition can have preconditions and postconditions, and the data it requires can be specified by input patterns and output patterns . For each component, the behavioral view is represented in state charts for each component, for example using state transition diagrams.

Hardware/Software Level / Technical Architecture Level

On the Hardware/Software Level or Technical Architecture Level (bottom level in Fig. 2.7), the artifacts are refined and enriched with realization information about the technical platform and the deployment. The Context artifacts are the Operational Context (SW) and the Technical Constraints (left column in Fig. 2.7), the Requirements artifacts are the Functional Requirements (SW/HW) (middle column in Fig. 2.7), and the Design artifacts are the Component Model (SW), the Component Model (HW), and the Deployment Model (right column in Fig. 2.7).

Operational Context (SW). The operational context on the software / hardware layer documents the operational environment of the software components to be developed. This refinement includes measures and sensors as well as controlled variables and actuators.

Technical Constraints. The technical constraints contain a compilation of all relevant technical constraints. This applies for example to external interfaces that have to be adhered to, and restrictions with regard to the software and hardware design of the system.

Functional Requirements (SW/HW). The functional requirements describe the requirements for the software of the planned system in form of a function model that specifies the functions in detail. They are described using templates. They refine the function requirements of the function group layer.

Component Model (SW). The software model depicts the technical software components, also called *clusters* and their behavior during execution. A runtime view represents the system behavior as interaction of clusters, events, timers, and buffers. For each cluster, an event is defined as activating condition, whose occurrence activates the cluster and makes it runnable. On each hardware component, there can only be one running cluster at the same time. Therefore, priorities have to be defined when there exists more than one runnable cluster simultaneously. An event can be based either on an incident in the environment or on an incident inside the system, for example the update of a buffer.

Realtime systems are time-critical and these specifics are modeled through timers. The expiration of a timer can trigger an event and activate a cluster. Thereby timers and events can model time-controlled processes. The communication between clusters is realized asynchronously with buffers. All modeling elements can feature hardware-independent, hardware-dependent, and context-dependent characteristics (for more detail, see [Wiled]).

The software model is represented in an execution view with graphical elements for the described elements cluster, event, timer, and buffer. This view also allows to depict the operational constraints that were still missing on the logical level.

Component Model (HW). The hardware view describes the topology of the hardware parts of the platform. All real hardware parts are abstracted to appropriate hardware components and their most important characteristics are specified. As the aim is to develop software for embedded systems, only those characteristics are of interest that directly influence the behavior of the software. Such characteristics are, for example, for communication busses the transmission times, for sensors the update rate, and for micro-controllers the size of the available data storage.

The notation may, for instance, be a box-and-line diagram depicting the topology of the hardware components with all ECUs, sensors and actors, and communication devices (data busses and cables), each of them attributed with their specific relevant characteristics.

Deployment Model. The deployment model describes the mapping of software to hardware. It maps the software elements cluster, event, timer, and buffer to their respective hardware components.

The notation is either a graphical mapping between the elements of the two diagrams for the software model and the hardware model, or a table that assigns the software components to hardware components, optionally with graphical support.

Relations between the Artifacts

This section explains the general relations between the artifacts.

The system vision and the stakeholder model build the basis for the definition of goals and general conditions for the planned system, the operational context and its development.

Goals and general constraints are the starting point for the goal model, the refinement and modeling of use cases, as well as the derivation of detailed design constraints of the operational context.

Use cases are the vital analysis and modeling technique for the derivation of necessary usage functions of the system and their behavioral specification via a function net and the adequate data models and interface specifications. They serve for the definition of roles and interfaces for external actors and components in the operational context on the function group layer and, furthermore, for the derivation of function requirements.

The conditions of the operational context, in combination with the function requirements and the function net, determine the logical architecture of the component model and its behavioral specification.

The technical constraints influence the derivation of the hardware components (the topology) and the software components from the logical component model of the function group layer. The component model for the software is enriched with a behavioral view through the help of the behavioral model of the function group layer under consideration of the technical constraints and the function requirements.

The deployment model maps the software components of the component model to the hardware topology of the component model hardware, again under consideration of the technical constraints.

Discussion Aspects

The above definition of the artifact model leads to a number of discussion aspects that have been analyzed and are dealt with in the following. These issues are the distinction of logical and technical description of a system, the quality of the artifacts, possible tailoring, the assignment of artifacts to the responsible originators, and the mapping of the artifact model onto document structures.

Logical vs. Technical Description. Worth discussion is the difference between logical (components and connectors) and technical (signals, bus communication, . . .) description of the system and in what case which level of abstraction is more suitable.

In the automotive domain, requirements specification documents often consist of a rather unstructured mixture of logical and technical descriptions. For example, such a document begins with a logical description in the form of use cases and provides some quality requirements, but then skips the logical architecture level completely and instead directly presents a technical architecture with a lot of detail, that on the one hand does neither explicitly relate to the requirements presented before nor give any rationale for the design, on the other hand with details of constraints that did not occur in any form previously in the document.

Where does this information come from? Many of the embedded systems in the automotive domain have been developed over and over again, for example within product lines. Therefore, an experienced developer will know the majority of the implied constraints without bothering to capture them in early artifacts like the Operational Environment. Furthermore, it is not necessary to reinvent the wheel every time there is a new release of an old system planned. Therefore, the technical architecture is already predefined, at least to a certain extent. This leads to the shortcut of the technical architecture being copied and skipping the development of a logical architecture. Some OEMs now use a new shortcut, by eliciting requirements as recommended by the state of the art, then still copying the technical architecture, and afterwards trying to link the two via defining a logical architecture. This already leads to better traceability but does not bring the main benefit of model-based development.

The development of a logical architecture and the explicit modeling of a system's behavior on a solely logical abstraction level allows for better quality through early model-checking or simulation and therefore early troubleshooting. Of course, the realization then does require the proper modeling of hardware and operational constraints on a lower abstraction level to be able to simulate the software and perform software-in-the-loop tests before deploying on real hardware, which is much more costly. Therefore, the answer to the question "Use logical description or technical description?" is: "Both."

Quality of the Artifacts. If the effort is taken to develop the artifacts described above, it is crucial to develop these artifacts in an adequate quality for really achieving the benefits of model-based development. Being able to audit the quality of the artifacts requires suitable metrics for measuring the quality.

A simple way of providing support for a certain artifact quality is to use criteria that define what content has to be specified to which degree of detail by a certain artifact. This approach was chosen, for example, for the REMsES guide [RDSS09].

Another approach is to develop the artifacts with sufficient quality by adhering to a number of common requirements quality criteria, as for example given by Robertson and Robertson [RR07]:

- Completeness
- Traceability
- Correctness
- Unambiguity
- Understandability
- Consistency
- Testability
- Atomicity

This list of criteria is commonly accepted in the RE community as other book authors in the requirements engineering discipline frequently cite these criteria, for example Pohl [Poh07]. Recknagel and Rupp have also proposed to use metrics for these quality criteria to provide quality assurance in requirements documents [RR06a].

By setting up metrics for the quality criteria, it is possible to define quality goals for a requirements specification based on quality goals for single artifacts. Such metrics for quality criteria are also used by process maturity models like CMMI [Sof09] and SPICE [SPI05] to assess the quality of documentation.

For defining appropriate quality assurance for the artifact model, it is reasonable to also consider tailoring of the artifact model.

Tailoring. According to the project situation, the artifact model and the related development process can be customized. In general, there are three types of tailoring as described, for example, by Kuhrmann [Kuh08]:

- According to characteristics of the organization of the developing company or association of developing companies, for example to fit standardized development processes.
- According to characteristics of the project type, for example single system development for customers, or product development for the market.
- According to characteristics of the process instance, for example the system domain, or the degree of criticality of the system.

This order provides three stages of tailoring, although they can be performed by the same person or instance. The bigger the company, the bigger the differentiation that occurs between the single stages. According to the size of the company, such an artifact model will be used company-wide or in single departments. The person responsible for the process first makes adjustments to the general process standards of the company, for example in terms of vocabulary that is preallocated within the company, or procedures for coordinating artifact contents. The result is a process conform to the general company processes.

In the second stage, the processes are adapted to the project type present, that is to the known initial situation and point of departure of the project. This stage links the processes with the preceding and following processes via clearly defined interfaces, for example in form of milestones with deliverables and acceptance criteria, depending on whether and in which form information is available in advance and which stakeholders are involved. Thereby, the optional process parts are appointed and the artifact model is tailored. The result is a reference for all projects that fit the defined initial situation. It should be communicated company-wide and be used throughout all projects of a similar point of departure.

Furthermore, it can be necessary to carry out further tailoring for special projects. In the third stage, the person responsible for the process adjusts the artifact model and process with respect to special requirements of the project. This is done in coordination and accordance with the project manager to special requirements of the project present. Regarding the process this may, for example, be a certain course of action in the collaboration with a subcontractor, regarding the artifact model this may be an amendment of voice recordings of stakeholder interviews.

The REMsES artifact model allows for all three stages of tailoring but does not describe a process how to exactly perform the tailoring. In general, the artifact model may be pruned and notation techniques for the individual artifacts may be chosen.

Mapping the Artifacts onto a Document Structure. The exchange of requirements for software development is still document-based in most cases. When presenting an artifact model, it is therefore important to distinguish between the structured system model, represented in different views by the artifact model, and the document structure that contains the artifact model, or parts of it.

A possible mapping of the artifacts to the prototype outline of the IEEE Recommended Practice for Software Requirements Specification (SRS) [IEE98] is presented in Tab. 2.5.3. The design artifacts of the logical and technical architecture levels are not included in the table as they do not fit into the defined general scope of the IEEE definition of a SRS[15] but are intended as connection to the subsequent design phase.

The intention of the mapping is to show that the artifact model can be fit into existing document structures, and does not require the introduction of completely new organizational structures for requirements engineering documentation.

Artifact Responsibility. There are usually different roles that produce the artifacts described in the sections above, for example a marketing manager, a requirements engineer, a system architect, and a quality assurance person. This differentiation is not of explicit interest for this work, the only two roles that are

[15]"The SRS writer(s) should avoid placing (...) design(...) in the SRS." [IEE98, p.3]

Table 2.2: Mapping of Artifacts to the IEEE SRS Prototype Outline [IEE98]

IEEE SRS Prototype Table of Contents	REMsES Artifacts (or *comment*, where not applicable)
1. Introduction	
1.1 Purpose	*purpose of the SRS, intended audience*
1.2 Scope	System Vision, Goals, Stakeholders
1.3 Definitions, acronyms, abbreviations	*as glossary or in form of references*
1.4 References	*list of documents (title, date, publishing organization)*
1.5 Overview	*outline and organization of the rest of the SRS*
2. Overall description	
2.1 Product perspective	Operational Environment, Interface Specification
2.2 Product functions	Use Cases, Service Graph, Behavior Specification
2.3 User characteristics	*reference to*Ê Stakeholders
2.4 Constraints	Technical Constraints
2.5 Assumptions and dependencies	*references to*Ê Operational Environment
3. Specific requirements	Functional Requirements, Quality Requirements

used for task descriptions in subsequent chapter are the requirements engineer and the system architect, as defined in the motivation (Sec. 1.1). However, the special case of distributed development imposes the need to differentiate which artifacts are provided by the OEM and which are produced by the subcontractor.

This is also closely related to the compatibility issue, as the artifact model has to be used by both the OEM and the subcontractor. The artifact model is a logical product data model and to use it, a concrete instance has to be defined. Interoperability can most easily be guaranteed, if both OEM and subcontractor use the same concrete instance of the artifact model. Due to the OEM cooperating with a multitude of subcontractors and each subcontractor usually working for a multitude of OEMs, this is hardly possible on a big scale. Therefore, the artifact model in this work can be instantiated with different description techniques and notation methods for compatible instances of a product data model, thereby minimizing integration problems.

There are two possibilities for artifact authorship according to which abstraction level was chosen for the subsystem assignment: on the logical architecture level or on the technical architecture level. In case of the assignment of a logical subsystem, the OEM would provide the artifacts defined on the usage level as well as any existing context and requirements artifacts on the lower levels related to that special subsystem. In case of the assignment of a technical subsystem, the OEM might already make prescriptions about the logical architecture and provide the respective specifications. The missing artifacts have to be produced by the subcontractor. In most cases, the exchange will be document-based.

Tooling

An artifact model like the REMsES model will only be used in practice if sufficient tool support is available. There are a number of possibilities to support the usage of the artifact model with adequate tooling, either captured in one tool as good as possible, or across more than one tool with an adequate management, for example through a controlled versioning system.

CASE-Tools. A simple solution is to tailor the artifact model in such a way that it can be represented completely with one tool. Common CASE-tools frequently used in the industry are Enterprise Architect[16], Rational Rose[17], Together[18], Visual Paradigm[19], and Simulink[20]. Apart from the last one, all of these tools are based on UML. Either of the UML tools can be used to produce the artifacts of the model.

Simulink was developed as tool for model-based software development for embedded systems but intended on a rather technical level building on Matlab[21]. Simulink may be used for only a subset of the artifacts as the context and requirements artifacts are out of scope for the provided modeling support.

In contrast, AutoFocus2[22] in combination with AutoRaid [SFGP05] is a prototype tool suite developed at TUM Software & Systems Engineering [FFH$^+$09a, SFGP05] which supports a tight integration of requirements engineering and system design. It allows for modeling with defined semantics across the abstraction levels defined for the artifact model. All artifacts can be modeled within AutoFocus2 and AutoRaid with consistency checks and traceability between requirements and design, including explicit rationale for the derivation of logical components.

REMsES tool concept. There are tools for developing the single artifacts, but there is no adequate tool support yet for managing a heterogeneous artifact model with relations between the artifacts. To fill this gap, a tool concept is proposed that supports the artifact model with a defined workflow for consistency and quality checks with respect to the content of and relations between the artifacts.

The REMsES tool prototype [PBP09] is a cross-tool solution that links the different artifacts via an additional graphical model representing the artifact model. Technically, it integrates a version control system with a ticket system and triggers automatic checks. It offers automatic consistency checks where this is possible and assigns tickets to reviewers for manual checks where automatic checking cannot be supported. It offers a lightweight, adaptable, workflow-centric support.

For each artifact there are four generic steps: Automatic content check, manual content check, automatic relation check, and manual relation check. Fig. 2.8 shows the states each artifact passes. In case of a failed check at any stage, a ticket is issued to the responsible developer and the artifact moves back into the state *work*.

Work: This is the first state of each revision of an artifact. If the artifact is in the state *work*, the author can make any changes without triggered effects.

Check Content Automatically: The state *check content automatically* is the starting point of the quality management system. If the developer

[16]`http://www.sparxsystems.com/`
[17]`http://www-01.ibm.com/software/awdtools/developer/rose/index.html`
[18]`http://www.borland.com/us/products/together/index.html`
[19]`www.visual-paradigm.com/`
[20]`http://www.mathworks.com/products/simulink/index.html?ref=pfo`
[21]`http://www.mathworks.com/products/matlab/index.html?ref=pfo`
[22]`http://www4.in.tum.de/~af2/`

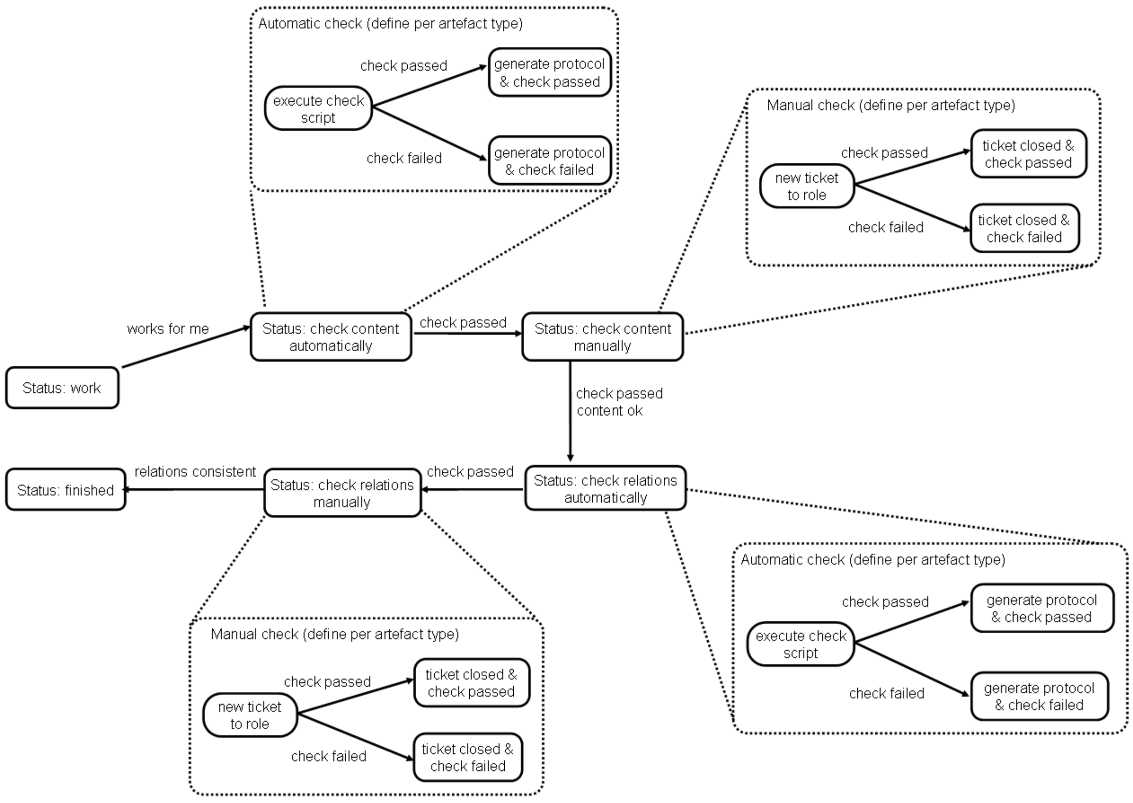

Automatic check (define per artefact type)
execute check script
check passed
generate protocol & check passed
check failed
generate protocol & check failed
Manual check (define per artefact type)
new ticket to role
check passed
ticket closed & check passed
check failed
ticket closed & check failed
Status: work
works for me
Status: check content automatically
check passed
Status: check content manually
check passed content ok
Status: check relations automatically
check passed
Status: check relations manually
relations consistent
Status: finished
Manual check (define per artefact type)
new ticket to role
check passed
ticket closed & check passed
check failed
ticket closed & check failed
Automatic check (define per artefact type)
execute check script
check passed
generate protocol & check passed
check failed
generate protocol & check failed

Figure 2.8: Generic Workflow of the REMsES Tool Prototype

sets the state of an artifact to *check content automatically*, the predefined content checks are executed. This includes all formal checks that can be conducted automatically. If these are passed successfully, the artifact receives the state *check content manually.*

Check Content Manually: In this state, the developer responsible for the review gets a ticket issued to perform the manual content checks, for example, to validate consistency with the initial requirements. If these checks are passed, the artifact receives the state *check relations automatically.*

Check Relations Automatically: The next step is an examination of the artifact in relation to other artifacts, i.e., the validity of each relation. These relations are defined in the artifact model. In this state, the predefined scripts for the rules that can be checked automatically are executed. If the artifact passes all checks, it moves to state *check relations manually* and the responsible reviewer gets notified.

Check Relations Manually: In this state, the remaining manual checks for relations have to be performed. If an artifact passes every check, it moves to state *finished.*

Finished: An artifact with the state *finished* has passed all the checks concerning its content and its relations to other artifacts.

The state of the whole artifact model can be determined from the current states of the contained artifacts and their relations. More concrete, the artifact model is said to be consistent if the current version of each artifact is in the state *finished*, meaning that it has passed all consistency checks. The aim of the supporting tool is to guide the users systematically to a consistent artifact model.

The architectural concept for the REMsES tool is based on a version control system (VCS), with a specific structure, which is complemented by an issue tracking system. The VCS serves as a central database and the issue tracking system provides the possibility to integrate the responsible developers into the workflow.

The proposed concept was realized on the basis of the two freely available tools: Subversion[23] as VCS and Trac[24] as issue tracking system. The prototype serves as proof of concept, but is currently not further developed.[25]

Traceability. In the case when one of the tooling approaches from above is in use, traceability is often provided by the tools, either because it is a CASE-tool that is based on an underlying consistent model, or because traceability was added to a cross-tool solution in terms of an additional graphical model representing the artifact model.

In the case when no specific tool support is defined, traceability has to be provided manually by the developers. This includes continuous updating of references, which is a tedious task that is likely to be forgotten or neglected. Therefore it is strongly suggested to define appropriate tool support when adopting the artifact model.

2.5.4 Results and Evaluation

The REMsES guide for systematic requirements engineering and management was evaluated in several case studies, student experiments, and pilot projects at the sites of the industrial partners. The guide is now available for free download [RDSS09].

2.6 Example: Driver Assistance Systems

There were nearly 50 million cars on the road in 2008 [Sta09]. Half of EU citizens (50%) drive between 5,000 and 15,000 km per year [Eur06] and safety is the most important factor (54%) that EU citizens would take into account if they were to buy a car (but most would consider fuel consumption in parallel).

Active Safety. Apart from passive safety through airbags and crush zones, there is a lot of potential in active safety systems, also known as advanced driver assistance systems (DAS). The latter are a variety of independent electronic systems designed to help the driver maneuver through demanding

[23]http://www.subversion.org

[24]http://trac.edgewall.com

[25]The REMsES tool prototype was presented to the project partners and is documented in [PBP09], but it is not stable enough to be provided for public download and trial.

traffic situations. According to Lindgren [LCJZ08], their overall aim is to reduce traffic accidents and to make the driving experience easier and more efficient.

Driver assistance systems demonstrably decrease the risk of accidents [Deu09] and therefore, the automotive subdomain concerned with driver assistance has emerged as important development area for market competition. Some challenges concerning driver assistance systems, like driver data collection, design guidelines, and traffic impact, have been investigated in the German research project INVENT [VB06]. Examples for driver assistance systems were already given in Sec. 2.6.

Requirements Elicitation. Eliciting requirements for such a system imposes a number of questions to be answered beforehand, for example whether to create systems based only on formal rules and legislations or if drivers should be allowed to break traffic rules and still get assistance from the systems [LCJZ08].

Furthermore it is necessary to evaluate when, why, and how the driver likes to have assistance, because as Werneke et al. report, drivers do enjoy driver assistance systems for safety and comfort but still want flexibility [WKV08]. Additionally, according to Lindgren et al., cultural differences have to be taken into account as well [LCJZ08]. One approach for eliciting requirements for driver assistance systems is an in-depth analysis of accidents [BV06, BDS08].

Running Example: DAS plus two Subsystems

The subdomain of the automotive domain that was chosen for the examples throughout this work is the domain of driver assistance. Driver assistance systems support the driver in the driving process. Their aim is to increase car safety and, more generally, road safety. Examples of such systems are in-car navigation system with typically global positioning system (GPS) and traffic message channel (TMC) for providing up-to-date traffic information, adaptive cruise control (ACC), lane departure warning, traffic sign recognition, collision warning system, night vision, blind spot detection and driver drowsiness detection.

The examples used in this thesis are the “Radio Frequency Warner” and the “Adaptive Cruise Control”. The requirements and/or system specifications of both systems stem from real industrial specification documents.

Radio Frequency Warner (RFW)

The “Radio Frequency Warning” System is a driver assistance system that supports the driver in coping with the information flood in road traffic with the help of radio frequency signals for traffic sign recognition. It is a fictitious system that has been specified with the standard specification and documentation techniques by project partner Daimler as case study for the REMsES project [RDSS09].

The system vision of the RFW is that with the permanent increase of traffic and traffic-related information during the last decades, it is now quite likely for the driver to miss a sign when for example distracted or when a sign is covered by another car. The RFW system filters incoming signals and displays the relevant ones on the instrument cluster display for as long as they are valid. As integrated comfort function along with cruise control, it also produces alert signals when the car is too fast for the current speed limit. The complete requirements specification of the RFW system [Ris07] cannot be supplied within this document due to reasons of secrecy but examples are taken as excerpts.

Adaptive Cruise Control (ACC)

The driver assistance system "Adaptive Cruise Control" is an intelligent speed control system that automatically maintains a pre-defined minimum distance to the car in front. ACC systems use either a radar or laser setup to allow the car to slow when approaching another car and accelerate again to the preset speed when traffic allows. This is achieved through a headway sensor, digital signal processor and longitudinal controller. The example system specification used in this work also features Pre-Crash Safety (PCS), which means it warns the driver and/or provides brake support if there is an increased risk of a collision.

The original requirements that have been used in the ACC case study are documented in [FFH+09b].

Summary. This chapter explained this work's background: the Architecture Model (Sec. 2.3), the Requirements Engineering Reference Model (Sec. 2.4), Automotive Software Development (Sec. 2.1), presented the conducted study about the state of practice (Sec. 2.2), and introduced the REMsES project (Sec. 2.5). This knowledge was used as a basis for the approach presented in this work, which will be detailed in the following, starting with the reference catalogue of decomposition criteria in the next chapter (Chap. 3).

Chapter 3

Decomposition Criteria

Contents

The decomposition of a system is the first step into the direction of defining the architecture after the analysis of the requirements. For actually performing such a decomposition, it has to be analyzed *why* a system is decomposed, *what criteria* have to be considered and *how* to apply the criteria.

This chapter presents the theory and analysis of the decomposition of systems into subsystems with the objective to show the connection of the approach in this work to architecture design. A special focus lies on reuse for being a promising means to significantly lower development costs.

First, related work for system decomposition is discussed. In Sec. 3.2, the factors that shall be optimized by state-of-the-art system development, which will be called "optimization factors" in the following, are introduced. Then, the criteria that have to be taken into account for system decomposition are presented in the "Criteria Catalogue" and its subcategories in Sec. 3.3, Sec. 3.4, Sec. 3.5, and Sec. 3.6, respectively. Subsequently, the coherence of the criteria is described in Sec. 3.7 and, finally, their impact on the decomposition (Sec. 3.8).

3.1 Related Work for the Decomposition of Systems

In the area of the decomposition of systems, there are two major approaches: general paradigms and practical guidance with patterns. Furthermore, there are collections of best practices.

General Paradigms. General paradigms are principles that support the task of decomposing a system. The two most cited ones are hierarchical structuring and information hiding. These paradigms were identified and presented in various papers by Parnas and Dijkstra: Parnas discusses criteria to be used in decomposing systems and promotes the concept of *information hiding* [Par72], extends this discussion for the development of program families with stepwise refinement [Par76]. Furthermore, he presents design decisions for software that is subject to frequent changes and therefore faces extension and contraction problems [Par79]. Dijkstra discusses the concepts of sequential processes and hierarchical abstraction levels [Dij68]. Another important paradigm is "Conway's Law" [Con68] (later extended by Herbsleb and Grinter [HG99]) which states that the structure of an organization is mirrored by the structure of the systems developed by the organization.

These general paradigms are an important foundation for the reference criteria catalogue.

Formal Approaches. There are also theoretic approaches on formal decomposition, for example as performed by Abadí and Lamport. They analyze the formal decomposition of a system in their paper "Decomposing Specifications of Concurrent Systems" [AL94] on which they base a formal conjunction of component specifications for program verification. The aim of their work is the decomposition of an existing system with the intent of performing program verification, which is not in the scope of this work.

Guidance and Patterns. One of the best known books on design and software architecture stems from the Pittsburgh Carnegie Mellon University Software Engineering Institute (CMU SEI), "Software Architecture in Practice" from Bass et al. [BCK03].

The presented approach relies on reference models, tactics, and architectural patterns. It uses knowledge from the attribute-based architectural styles (ABAS), where the idea of object-oriented design patterns is applied to architecture [KK99]. Furthermore, it adapts the Attribute-Based Design (ABD) method [BBC$^+$00]. Clements worked on the transition of domain models to architecture [Cle94].

The successor method for ABD and most recent development from the CMU SEI is Attribute-Driven Design (ADD) [WBB$^+$06]. In ADD, the design process for architecture relies on quality attribute requirements. Inputs are functional requirements, design constraints, quality attribute requirements, implied constraints, and uncategorized requirements (e.g., related to legacy systems). The process identifies architectural drivers and applies design concepts (patterns). Outputs are software elements, roles, responsibilities, properties, and relationships.

All of these approaches rely on patterns that match architectural drivers, which reflect only part of the decomposition criteria given in this work. DeSyRe instead emphasizes on an analysis of decomposition criteria *before* such a pattern matching. Thereby, the approach takes into consideration the whole range of potentially influencing factors.

Best Practice. Accumulated knowledge from best practice is presented by Hofmeister et al. in their book "Applied Software Architecture" [HNS00]. They organize relevant architectural factors in three categories, namely *product factors* (e.g., functional features, user interface, performance), *technological factors* (e.g., software technology, architecture technology, standards) and *organizational factors* (e.g., management, staffing, development schedule), and then use Kruchten's 4 views [Kru95], i.e., conceptual, module, code, and execution view.

The approach first prescribes a global analysis of factors, where the architects describe, characterize changeability, analyze impact, and sum up the gained knowledge in factor tables. Then, the authors suggest to develop so-called *strategies* by identifying an issue, developing a solution, identifying related strategies, and summing up in issue cards. From those, central design tasks are derived.

The idea of the book is to sum up best practices and make them easily available to other practitioners. The categorization of criteria mixes factors from different stakeholders while the DeSyRe categorization distinguishes between factors according to their source of origin and therefore provides support for gathering the respective information during requirements engineering.

3.2 Overview of the Criteria Catalogue

The decomposition process is influenced by system type specific aspects as well as domain specific and individual constraints. The criteria catalogue presented in this section reflects both points of view in four criteria categories that guide through the decomposition process. As stated in Sec. 3.1, up to now, there is no encompassing catalogue of decomposition criteria in literature.

This section first explains the objective for the decomposition of a system, i.e. the envisioned optimization, subsequently gives an overview of the criteria categories, and introduces the description template that is used to explain each criterion in the following sections. The subsequent sections then discuss the criteria of each category in detail and illustrate them with examples.

The catalogue was developed from a number of resources: First of all, an extensive literature research, inter alia Conway [Con68] and Herbsleb [HG99], Nuseibeh [Nus01], Parnas [Par72], and Wojcik [WBB$^+$06].

Further information sources were requirements engineering activities and related reference models (for example REM, see Sec. 2.4 and REMsES, see Sec. 2.5). The gained knowledge was validated and enriched within the interviews of the field study (see Sec. 2.2) and, last but not least, through many discussions with colleagues from our research group and other institutions.

3.2.1 Optimization Factors

Decomposition of a system takes place with a certain aim of optimizing the development. The commonly accepted basis for optimizing development and projects in general is the triangle of time, quality and costs [MHM87], depicted in Fig. 3.1 that needs to be balanced.

Further optimization factors can be related to one or more of these three major concerns, for example, related to costs and quality is the issue of reuse,

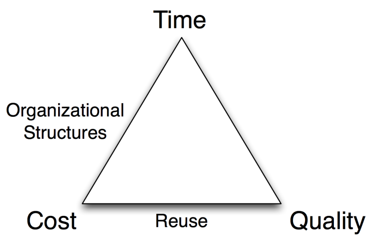


Figure 3.1: Time-Cost-Quality Triangle and Related Concerns.

and related to time and costs are organizational structures. There may be other potential issues for optimization but these are the ones considered important within this work.

3.2.2 Criteria Categories

The four categories are the *directive criteria*, the *functional criteria*, the *quality criteria*, and the *technical criteria*. Each of the categories is related to different stakeholders, as visualized in Fig. 3.2.

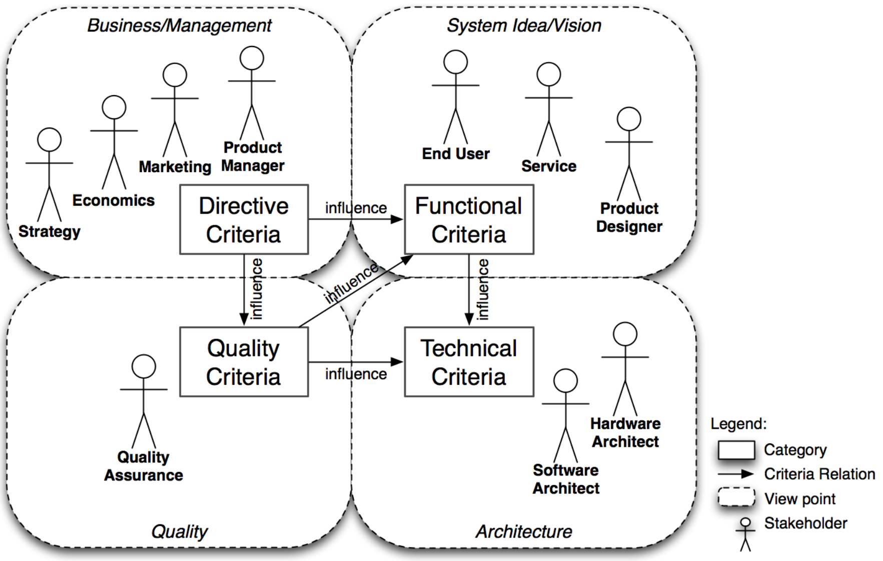


Figure 3.2: Decomposition Criteria Categories and their Stakeholders

Directive Criteria: The stakeholders of the *business* point of view are the strategy consultant (or business analyst), the economist, the marketing expert, and the product manager. They issue the majority of directive criteria. The *directive* criteria contain laws and standards, licensing/patents, information politics and business rules.

Functional Criteria: The point of view of end users, service personnel, and product designer is the *system idea*. These stakeholders provide the functional criteria and (indirectly, therefore not depicted in Fig. 3.2) part of the quality criteria. The *functional* criteria are concerned with the usage functions the system shall provide.

Quality Criteria: The point of view of *quality* is represented by the quality assurance manager. The *quality* criteria reflect implications by desired quality characteristics of the system.

Technical Criteria: The stakeholders who hold the *architecture* point of view are the software architect and the hardware architect. The *technical* criteria derive from the constraints given by the technical solution domain and the future system environment as well as from architecture design rules.

Thereby, not all categories have to be present to the same extent for a system, for example a web application would probably have to conform to certain business rules that are of less interest for an embedded system.

Due to the aforementioned dependency of the criteria on the type of system and its specific situation, there is no general rule on how to incorporate all the criteria equally into the decision for the division into subsystems. The relevance of the criteria has to be evaluated separately for each system and according to that relevance, the division can be decided individually. It is not desired to limit the general applicability of the approach but, to be able to justify the selection of criteria for the division, it is necessary to assume a certain type of system in the following. The catalogue of criteria will be the same but the criteria's priorities might vary for specific systems or types of systems.

The type of the specified system determines to what extent each of them is relevant for its division into subsystems. In this work, it was already defined that the system type of the given context is embedded systems.

3.2.3 The Description Template

The template used for explaining each criterion in the remainder of this chapter was created to provide an easy overview and systematic description of the criteria of the different categories.

Table 3.1: Description Template for Decomposition Criteria.

Template entity	*Description of the entity, to be filled in for each criterion*
Source	<Corresponding documents for information retrieval>
Impact	<Priorities, consequences and risks>
Usage	<Recommendation of state-of-the-art methods>
Examples	<From case study scenario “international logistics company”>
Prioritization	<According to the reasons for decomposition, according to the business domain, and according to the system type>

The template for the criteria description is depicted in Tab. 3.1. It includes the *source of information*, which explains where the information for this criterion can be gained, the criterion's *impact* or priority, which describes what can happen if it is ignored for the decisions, the *usage concepts*, which give guidance on how to proceed, and *examples*, which have been chosen from different case studies. There have been identified three different “dimensions” for the *prioritization* of the criteria for the decomposition of a certain system:

- according to the intention (reasons for performing decomposition, e.g., control of complexity, improvement of reuse, . . .),
- according to the business domain (e.g., aerospace, financial sector, consumer electronics),
- and according to the system type (e.g., BIS / data-intensive, embedded system / performance-intensive, multimedia & entertainment / performance & data-intensive but uncritical).

Due to a limited generality of the criteria, it is only possible to give hints and guidance for the possible consequences for the architecture and concrete usage concepts. The usage concepts will be methods in the small that enable simple application of the approach to ensure acceptance in practice.

The sources of information for the criteria within the requirements and design artifact model used throughout this work (explained in Chap. 2.5.3) is depicted in Fig. 3.3.

The figure is intended to provide a rough idea about which part of the artifact model the criteria may be derived from, according to content category and abstraction level (both concepts were introduced in Sec. 2.5). The directive criteria have their origin in the area of the context on the system level, the quality criteria stem mainly from the requirements on system level and function groups level, the functional criteria are emphasized in the design on the system level and the requirements on the function groups level, and the technical criteria are mainly found on the software / hardware level spread across the content categories context and requirements.

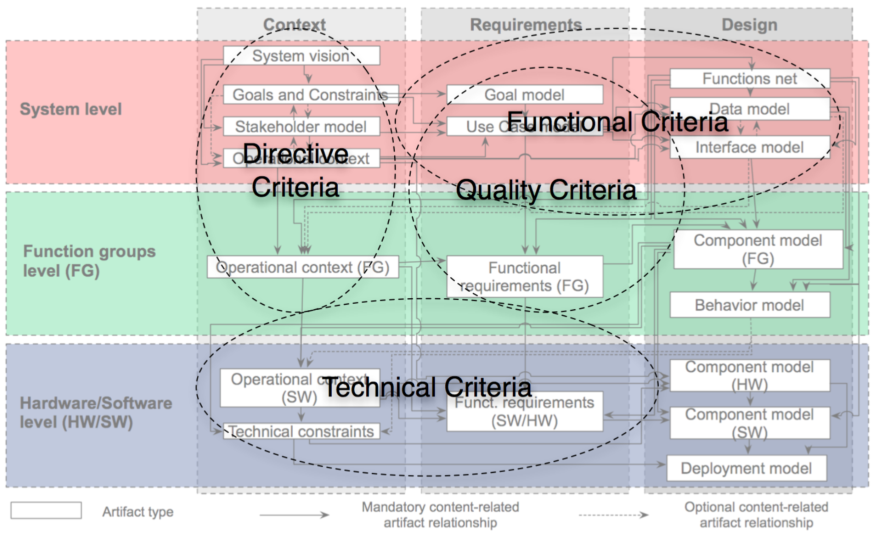


Figure 3.3: Criteria Information Sources within the Artifact Model

Each of the categories will be described in detail in the following sections and the criteria are illustrated with a continuous example. The template is filled for each criterion to provide the information flow and handling of the criteria during the requirements engineering process.

3.3 Directive Criteria

Depicted in the upper left corner of Fig. 3.2, the directive criteria summarize all criteria that influence the system's decomposition from "the outside", to say independent from the system vision itself. They can be derived from the perspective of the business (requirements), represented in the REM specific document type reflecting the *Business Needs* (see Sec. 2.4). The criteria encapsulate direct influences, which cannot be modified, and indirect influences, which are negotiable to a certain degree. They can be divided into organization, legislation, and economics. Bass et al. refer to some of the following criteria as "business qualities" whose "goals center on cost, schedule, market, and marketing considerations" [BCK03, p. 95].

The following list of directive criteria was elaborated through literature research, discussions with colleagues and project partners from industry, and validated by the study described in Sec. 2.2. The grouping into the three domains organization, legislation, and economics was chosen for easier overview and orientation.

- Organization
  - Infrastructure and Conway's Law
  - Information politics, business rules, and implications from subcontractor-supplier relationships
  - Experience, background and expertise of the developers
- Legislation
  - Laws
  - Standards
  - Licensing
  - Patents
  - Certificates
- Economics
  - Reuse
  - Cost models
  - Demand management

There is general consensus on that these factors do influence a system's design, but in part it is hard to put a finger on a concrete example, because those influences are indirect. Some aspects like experience and personal background may not even be perceived consciously as an influence by the system developers in charge.

3.3.1 Organization

Table 3.2: Organizational Criteria

Source	Business objectives, general conditions, scope
Impact	System acceptance criteria are not accomplished and can not be traced.
Usage	“Means” from non-functional requirements (NFR) method [DKK^{+}05], system concepts undergo an acceptance test before they are designed (technical solution variations and corresponding acceptance criteria for guiding through the test).
Examples	The coarse-grained decomposition has to be structured according to the development units for each special domain (e.g. driver assistance). No use of open source components (e.g. in military systems).
Prioritization	According to business domain and system type.

Information sources, impact, usage, example, and prioritization are described in the template in Tab. 3.2.

The influence of an organization’s *infrastructure* on the structure of a system has first been explored and described by Conway [Con68], then Parnas [Par72] and was later on revised by Herbsleb [HG99]. The following citation from the original work of Conway [Con68] concludes his essay with a result statement known as “Conway’s Law”:

> “...organizations which design systems (in the broad sense used here) are constrained to produce designs which are copies of the communication structures of these organizations.” [Con68, Conclusion]

Parnas shows that dividing a software system is simultaneously a division of labor as he defines: “In this context ’module’ is considered to be a responsibility assignment rather than a subprogram.” [Par72].

Herbsleb and Grinter showed that for distributed development, Conway’s Law does make sense, but focused on the problem that “multiple site development works against informal communication channels” and therefore complicates integration. This problem was also reported in the accomplished study interviews (see Sec. 2.2) as communication efforts between OEMs and suppliers and will be tackled in this work through the artifact model (Sec. 2.5.3).

Furthermore, the organizational issues capsule *business rules* and *information politics* that imply certain restrictions on the development process, for example human resource management. Implications from subcontractor and *supplier relationships* are agreements on the workflow intersection or interface, e. g. the type of reference document or specification that serves as contractual basis for the collaboration.

Finally, experience, background and expertise of the developers influence the design of a system as well. “If the architects of a system have had good results using a particular architectural approach, (...) chances are that they will try that same approach on a new development effort.” [BCK03, p. 8]. The same applies conversely for approaches that worked poorly.

3.3.2 Legislation

Table 3.3: Legislational Criteria

Source	Laws, limitations and scope
Impact	Road traffic concession
Usage	Checklists and guidelines for global software development [BH07]
Examples	"The recovery rate is defined and calculated according to ISO 22628." [Ris07, RFW176]
Prioritization	According to each tailoring dimension there are different laws that have to be obeyed

Legislation features the non-negotiable criteria as these are straight forward influences from laws, e. g. the data protection act or road traffic regulations. Information sources, impact, usage, example, and prioritization are described in Tab. 3.3.

Within the external influences, there are regulations and permissions. *Regulations* are standards that the decision makers want to obey, e. g. ISO standards. The *permissions* capsule part of the influences from product management, in explicit which licenses, patents or certificates are needed or used; also an important cost factor.

Relevant laws in the automotive domain are for example the Automobile Safety Act [Bun07], the Product Liability Law [Kul06], and the Electromagnetic Compatibility Law [Bun08d]. To point academic audience to the technical level on which OEMs are currently dealing with legislative constraints, the current state of the art for handling them at Daimler's is presented and discussed in [PL08].

In the information systems domain, the German ministry for security in information technology ("Bundesamt für Sicherheit in der Informationstechnik") imposes strict orders and regulations on data storage and processing to be conform to the German data protection act [Bun08a]. The data protection act is one of the many laws that differ considerably in each country. One possibility to handle such differences in legislation of different countries is to define a role model with different views for each country. In embedded systems, constraints of country-specific laws like the road traffic act can hardly be solved by such a view. Therefore, there are different versions of a car, for example "US", "Asia" and "ROW" ("rest of world").

3.3.3 Economics

Table 3.4: Economic Criteria

Source	Business objectives, market analysis, risks, ROI
Impact	Endangered ROI, predictions of risk analysis fail
Usage	Component-oriented development (for reuse), modularization into stand-alone subsystems for off-shoring (depending on risk models)
Examples	Product line approaches and distributed development in the automotive domain in general.
Prioritization	According to reasons for decomposition, e.g. reusability or distributed development. For example, for distributed development, the risks of off-shoring have to be considered.

Economics includes reuse, cost models, and demand management. Information sources, impact, usage, example, and prioritization are described in Tab. 3.4.

The *demand management* is concerned with market positioning and strategic goals (of the developers as well as the customers) regarding the delivery of "business goods", including timing constraints like time-to-market. The *cost models* are chosen according to the intended product placement and management.

For systematic *reuse*, the analysis must identify variations to anticipate changes and the design must be chosen for adaptability [DH]. Therefore, another criterion for the decomposition of a system is the stability of the requirements or how fast they might change; the keyword here is "software aging" [Par94]. As this can be regarded as an aspect of evolvability within the quality criteria, it was not put on the list separately.

3.3.4 Directive Criteria of the Running Example

The directive criteria for our running example Driver Assistance Systems are listed and discussed in Tab. 3.5.

Table 3.5: Directive Decomposition Criteria in DAS

Organization	Infrastructure	Conway's Law says that the system will mirror the developing organization's structure, but at the same time, developing organizations will also be structured according to the systems they develop. The departments at BMW's are already organized according to the different driver assistance systems.
	Business Rules	No explicit business rules are known of that would influence the decomposition of DAS.
	Experience	The experience of the developers is about ten years in average, their background is mainly mechanical and electrical engineering.
Legislation	Laws	German Road Traffic Act "StVO" [Bun07] , German Road Traffic Admission Act "StVZO" [Bun08c], Electro-magnetic Compatibility Act "EMV" [Bun08d] and others, but none of them explicitly influences the decomposition.
	Standards	Different German Industry Norms "DIN" apply, but none of them is relevant for the decomposition.
Economics	Reuse	All of the DAS have already been developed in the previous vehicle series, so reuse is strongly expected.
	Cost Models	Cost of the hardware parts is crucial as even cents sum up critically for the number of produced vehicles. Software development costs are considered less critical, but in this case is connected with reuse, as a decomposition for high reusability will decrease the costs for the next iteration.

3.4 Functional Criteria

Depicted in the upper right corner of Fig. 3.2, the functional criteria that were identified are:

- Clustering in services (functional features) according to user perception.
- Functional dependencies.
- Unwanted feature interaction (in other words: side effects).

The first black board sketch of a system's decomposition will usually be a functional one defined by the usage behavior. The proposal is a *clustering into services* that will be offered to the user.

3.4.1 Clustering According to Services

Table 3.6: Clustering according to Functional Features

Source	Scenarios, functional requirements
Impact	"Intuitive" and process-oriented, but no explicit account for quality requirements
Usage	Broy [BKM07], QUASAR enterprise [Sie02], SOA [BS06], Rittmann [Rit08b]
Examples	RFW Feature description: "The system stores information about a possible speed limit during a parking stop and presents it to the driver at system startup." [Ris07, RFW_SL-66] Functional requirement from Diagnostic Service Lane Change Warning: "The system warns the driver about risky lane changes and supports him / her during the execution with a probably necessary correcting reaction to avoid an impending collision." [Gyö08]
Prioritization	According to reason for decomposition (= optimization factors). There is a difference in modelling services for distributed development and for distributed delivery, as the latter has to include considerations about the future usage domains of the system.

A service describes functional characteristics of the system from a user's point of view [Bro05] and therefore shows a black-box view onto the system. For the clustering of functionality according to usage services, the information sources, impact, usage, example, and prioritization are described in Tab. 3.6.

A decomposition according to identified services makes sense because it probably comes closest to an intuitive decomposition (e.g. as performed by Rittmann [Rit08b]). This sketch has to be checked in terms of dependencies and interaction between the services.

In order not to overly simplify reality, it has to be mentioned that sophisticated functionality may require the cooperation of various functions that can be distributed across different hardware units.

3.4.2 Functional Dependencies

Functional dependency means that a service or functional feature can only perform if another feature is either active and has delivered input or is waiting for input, or if another feature is deactivated. The information sources, impact, usage, example, and prioritization are described in Tab. 3.7.

Table 3.7: Functional Dependencies

Source	Functional requirements, behavior specification
Impact	Prevent conflicts in concurrent usage and preserve causal order
Usage	Perform dependency analysis [Grü08] and [Rit08b]
Examples	RFW: A warning tone is generated for urgent signals only in combination with display of the signal. (derived from [Ris07, RFW_SL-71]) ACC: "If the parking assistant is activated while the ACC is active, the ACC is deactivated and the parking assistant functionality is executed." [Rit08a]
Prioritization	Always has to be considered.

The decomposition of the system is not a pure top-down process, because there can only be limited knowledge about interactions and dependencies as long as the system's architecture is not yet modeled to a certain detail. Therefore it can be necessary to do the decomposition in iterations and incorporate the feedback from the interaction analysis if it reveals certain new dependencies during the first modeling approach of the system.

3.4.3 Unwanted Feature Interaction

Table 3.8: Unwanted Feature Interaction

Source	Functional requirements, behavior specification, function net (see Sec. 2.5.3)
Impact	Quality assurance for first system sketch, early detection of unwanted feature interaction
Usage	Analysis of unwanted feature interaction [Grü08]
Examples	The door of the car is opened and the warm air causes the air conditioning to increase performance in order to cool down to the set temperature. The extra energy required for that purpose causes higher revolutions of the engine. The increased revolutions are misinterpreted as a driver stepping on the gas pedal who would thereby want the hand break released to be able to drive off. Therefore, an unwanted release of the automatic hand break occurs and the car starts rolling away. [Grü08]
Prioritization	Optional, additional to dependency analysis.

A completely different type of interaction is "unwanted feature interaction" which is unwanted side effects instead of intended cooperation between services. In order to verify that there is no such interaction, an analysis has to be performed, for example as proposed by Grünbauer [Grü08]. The information sources, impact, usage, example, and prioritization are described in Tab. 3.8.

These functional dependencies and interactions can only be identified at the level of the detailed system concept in terms of REM (Sec. 2.4), where a first sketch of the technical solution is developed, or within the design category of the system or usage level in REMsES (Sec. 2.5), in explicit represented by the artifact function net, which is described in the next chapter (see Sec. 2.5.3).

3.4.4 Functional Criteria of the Running Example

The functional criteria of the running example "Driver Assistance Systems" are listed and discussed in Tab. 3.9.

Table 3.9: Functional Decomposition Criteria in DAS

Usage Services	The usage services that are present in the use cases in Sec. 6.1.4 can be grouped according to their functionality. This is the most intuitive decomposition.
Functional Dependencies	For the use cases in Sec. 6.1.4 there are some functional dependencies. For example, use case 2 is also relevant for use case 5 in case of a vehicle in the blind spot when departing the lane.
Unwanted Feature Interaction	Interaction occurs in case of two DAS responding at the same time to a traffic situation. For example, the cruise control accelerates when the driver sets the indicator to pass another vehicle but at the same time the system detects a vehicle in the blind spot and therefore has to stop the acceleration to prevent a crash with the vehicle in front as the driver cannot change lanes. Therefore, the decomposition has to enable easy and fast communication between the different DAS.

3.5 Quality Criteria

The quality criteria are depicted in the lower left corner of Fig. 3.2. It is important to include non-functional requirements (NFR) in the decomposition process as “The scope of formal specification and analysis must be extended to cover non-functional requirements that play a prominent role in architectural design – such as performance, security, fault tolerance, accuracy, maintainability, etc” [vL00].

It is a challenge to pin down the quality criteria in a way that they can be pursued satisfyingly during development, just because this type of criteria often comes up by underspecified demands of the customer. A refinement of the term *non-functional* brings up general constraints (project constraints, political constraints, technical constraints as shown in [RR06b]) and quality criteria.

There are many different opinions on (product-oriented) quality criteria, their interpretation and treatment as well as on software quality in general. The current variety of quality reference models (e.g. [WD07]), taxonomies, and maturity degree models has yet to be unified and standardized. In this work, the classification of *quality* is according to ISO 9126 [Int01] being the international standard for the evaluation of software quality. The quality model established in the first part of the standard, ISO 9126-1, classifies software quality in a structured set of characteristics[1]:

- Functionality (Suitability, Accuracy, Interoperability, Compliance, Security)
- Reliability (Maturity, Recoverability, Fault Tolerance)
- Usability (Learnability, Understandability, Operability)
- Efficiency (Performance in terms of behavior and resources)
- Maintainability (Stability, Analyzability, Changeability, Testability)
- Portability (Installability, Replaceability, Adaptability, Conformance)

The ISO 9126-1 [Int01] also lists “Functionality” within the characteristics. For the decomposition criteria, functionality was already considered in Sec. 3.4 and is therefore omitted here apart from the aspect of security. Security may

[1]These aspects focus on the quality of the specifications, e.g. regarding structuredness or completeness, *and* the development processes.

influence the decomposition of a system and has not been considered before within the catalogue at hand.

Each of these characteristics is refineable to a certain (individual) level of measurable quality attributes (see e.g. Dörr et al. [DKK$^+$05]), but we do not list them all as the refinement process depends on the individual business domain. For example, within IT Service Management the emphasis often lies on the availability of IT services, which is a subcategory of reliability, while for embedded systems the focus often lies on safety and performance aspects.

> "It is critical to design each structure correctly because each is the key to a different quality attribute: ease of change, ease of extracting a subset, and increased parallelism and performance."[BCK03]

For our approach, we limit the perception and measurement of quality to meet the requirements specified by the customer within the contract specification.

Table 3.10: Quality Criteria

Source	Business needs, general conditions, design constraints, scenarios, quality requirements
Impact	Higher maintenance effort, difficult change management and defect detection, decrease in user satisfaction
Usage	Checklists within quality reviews, audits, modeling (relate to use cases and refine and bind to state automata), design patterns [GHJV95], programming paradigms [Par72], SOFTPIT [HMRR06], CONQAT [DJH$^+$08], ASPIRE [DKK$^+$05]
Examples	Code quality, but not mentioned in [Ris07] as this would rather be a company standard. "The handling has to be simple and intuitive." [Ris07, RFW_SL-69] "For the choice of material and parts it has to be considered that spare part deliveries have to be possible until 10 years after the end of serial production." [Ris07, RFW329]
Prioritization	All 3 dimensions imply emphasis on different quality aspects, e.g. for the system type a BIS will request high productivity while an embedded system focusses on maximum safety. For BIS the general view is rather about business processes realized as application domains, while for embedded systems the view is rather hardware driven, e.g. by ECU.

However, the crucial point for the impact of quality criteria on decomposition is the possibility of actively designing the decomposition to support a certain quality attribute. The information sources, impact, usage, example, and prioritization are described in Tab. 3.10. Prominent examples for design rules are programming paradigms and principles as promoted by [Par72] or design patterns [GHJV95].

Finally, Bass et al. [BCK03] give concrete guidance on how to enhance quality attributes in an architecture by using the adequate patterns and principles. For each quality attribute, they list so-called tactics, which is the term they use to summarize patterns and principles, to enhance the specific quality. If possible, they are additionally organized according to their appropriate application time during the development process, e.g. at design time, binding time, and run time. For example, design time tactics for usability are the patterns Model-View-Controller, Presentation-Abstraction-Control, etc [BCK03, p. 123].

3.5.1 Quality Criteria of the Running Example

The quality criteria of the running example "Driver Assistance Systems" are listed and discussed in Tab. 3.11.

Table 3.11: Quality Decomposition Criteria in DAS

Functionality	In the definition of the ISO 9126-1 [Int01], functionality includes suitability, accuracy, interoperability, compliance, and security. Naturally, these factors are important for DAS but no explicit impact on the decomposition of DAS could be determined.
Reliability	Maturity, Recoverability, and Fault Tolerance are crucial for all vehicles participating in road traffic. For system decomposition, this can mean that certain subsystems are grouped onto redundant electronic control units (to prevent harm in case of malfunction).
Efficiency	Performance is required in terms of behavior as for any real time system and performance in terms of resources is required to keep the size and costs of the hardware low.
Maintainability	Changeability and testability are important for reusability. This leads to a typical modular decomposition with high cohesion and loose coupling.

3.6 Technical Criteria

Depicted in the lower right corner of Fig. 3.2 are the technical criteria, which usually arise bottom-up from either the system context given by the surrounding environment or from constraints of the technical realization domain. Architecturally significant requirements often arise from quality attributes, volume of functionality, architecting for a family of related systems, choice of technologies, deployment, and operations [BBC$^+$06]. According to their influence on the architecture, they can be grouped into three subcategories:

- Communication requirements
- Technical constraints
- Legacy systems

These criteria are rather concerned with the system design than with the business logics. The type of communication and therefore the corresponding *communication requirements* depend mainly on the purpose of the system, e.g. for a real-time application we have to guarantee that all data arrives in time, while for multimedia it is more important that we have a consistent data rate and can transmit high data volumes. The *technical constraints* derive from the constraints given by the technical solution domain and the future system environment. There may exist *legacy systems* that have to be taken into account. With respect to decomposition such legacy systems are relevant if they are components of the system, not only communication partners.

The technical criteria are therefore not only influenced by the system type, but also by (requirements of) the business domain. While guiding through the decomposition process, this criteria category links the system specific aspects with the (individual) requirements given by the business domain. This supports the idea of an artifact model across the content categories from context to design, as it unifies both points of view along several artifact types during the requirements engineering process.

3.6.1 Communication Requirements

Table 3.12: Communication Requirements

Source	Behavioural requirements, standards (for system apects, not documented within system specification)
Impact	Transaction security, data consistency, technical adequacy
Usage	Consider for technical architecture, for example, on the SOA layer, see Sensoria [FLB06]
Examples	"The car is not equipped with RFID transmitters. Communication will only flow in one direction, from the mobile or fixed radio signals to the cars." [Ris07, RFW_SL-72] Information system: "The IP protocol has to be used."
Prioritization	According to system type, e.g. for a BIS we think about communication protocols in terms of messages, while for an embedded system we think about the bit patterns for certain sensor values.

The communication requirements got assigned high impact on decomposition in the study interviews in Sec. 2.2. The information sources, impact, usage, example, and prioritization are described in Tab. 3.12.

For example at Daimler, the communication matrix, which displays the messages that are sent between the ECUs, is currently one of the most important artifacts for decomposing and designing a new system. The matrix only presents the content which has to be communicated but does not give details on how to do this. Further communication requirements are therefore spread throughout the requirements specification, especially within behavioral requirements.

3.6.2 Technical Constraints

Table 3.13: Technical Constraints

Source	Design constraints
Impact	Technical adequacy
Usage	Consider for technical architecture within the design on the software / hardware layer [BCK03].
Examples	"The frequency (of the RFID tags) is 5,8GHz." [Ris07, RFW_SL-217] "The data load of a CAN message is 0 to 8 bytes." [Rob91]
Prioritization	For BIS there are top-down technical requirements that can be realized by different solutions, while embedded systems also have bottom-up hardware constraints from different engineering disciplines that cannot be negotiated but have to be obeyed.

The term "technical constraints" is very general but the only adequate one, as there is a wide variety of constraints from the surrounding technical context and the future system platform. The information sources, impact, usage, example, and prioritization are described in Tab. 3.13.

These constraints are spread throughout the requirements specification and all types of so-called "further applicable documents". The latter is usually a quite extensive collection of various related specifications and the only way through that accumulation of specifications is experience. According to research partners from industry, experienced developers know one half of those constraints by heart and know for another third where to look them up - the missing part is the one that frequently causes integration problems during development. This situation can be improved by using the approach proposed by this thesis.

3.6.3 Legacy Systems

Table 3.14: Legacy Systems

Source	General conditions that refer to an old specification of a legacy system.
Impact	Incompatibility
Usage	Reengineering methods, compatibility checks as performed by Koss [Kosed]
Examples	"The implementation of the old system was realized in C." "The system has to support the 2004 and 2008 editions of the sensor Z42."
Prioritization	Reengineering methods and compatibility checks are individual for each specification technique and programming language.

At a first glance, there are not as many legacy systems in the automotive domain as for example in information systems, because the systems are always newly developed. At a second glance, the specifications are often reused and only minimally changed, if the economic calculation is still the same. Furthermore, new product lines are often based on preceding ones. The information sources, impact, usage, example, and prioritization for legacy systems are described in Tab. 3.14.

Especially embedded systems advance quite quickly as the general development speed in terms of hardware evolution is pretty high. New hardware, with either new materials or more capacity in storage or performance requires new software to enhance the complete system.

3.6.4 Technical Criteria of the Running Example

The technical criteria of the running example "Driver Assistance Systems" are listed and discussed in Tab. 3.15.

Table 3.15: Technical Decomposition Criteria in DAS

Communication Requirements	Easy communication has to be possible for real time interaction between the different DAS and with the rest of the vehicle.
Technical Constraints	Hardware topology and resources imply a number of technical constraints that are important for the final deployment onto the technical architecture but not for the decomposition with respect to the logical architecture design.
Legacy Systems	There is the surrounding system environment, but no real legacy systems have to be taken into account, as the complete system is under development with every new vehicle series.

3.7 Coherence of the Criteria

Similar to the optimization factors in Sec. 3.2.1, the criteria are not independent of each other but instead closely interrelated.

3.7.1 Dependencies between Criteria

This section analyses the dependencies between the criteria of the given categories. An overview is depicted in Fig. 3.4.

Directive to Functional and Quality Criteria. The natural flow of influence begins with the directive criteria: The functional and quality criteria may not be derived from the directive criteria in a compulsive way, but there is definitely an influence from them as the system under development has to fit the general business goals and market strategy.

Legislature is reflected by constraints for certain quality characteristics, for example in terms of reliability, the quality criterion has to specify how the reliability required by legislation is guaranteed by the system, e.g. by redundancy in form of multiple provision of controllers.

Economics are represented by business goals. These may also include high-level goals about certain quality characteristics, or simply the general notion "the user expects high quality of our product" which has to be refined within the quality criteria in a measurable way, for example a high mean time to failure and comfortable usage.

Directive to Technical Criteria. Furthermore, the directive criteria influence the technical criteria: The legislation can influence or be the source of some technical constraints, like the motor vehicle certification act [Bun08c] or the United Nations Economic Commission for Europe R43 [Uni03], which for example specifies that the front window of a car may not be darkened below a translucency of 75%. The latter restriction may be used directly as technical constraint.

Economics can also be the source of origin for technical constraints, for example when a certain hardware has to be used that imposes a restriction on the reaction time but is cheaper. The result of this trade-off due to price calculations is a technical constraint with the reaction time of that particular hardware.

Economic calculations are also often the reason for using (parts of) legacy systems, because it is either simply cheaper or not possible to redevelop all required subsystems at the current point in time.

Use of legacy systems may also be due to organizational issues, for example, probably rather in business information systems than in embedded systems, a "that's the way it has always been" attitude. A more likely organizational reason for usage of legacy in embedded systems development are differing development cycles at some of the suppliers' sites that do not perfectly fit the OEM's development cycle. In that case, the term "legacy" refers to a system that is probably only two years old, but already not any more state of the art.

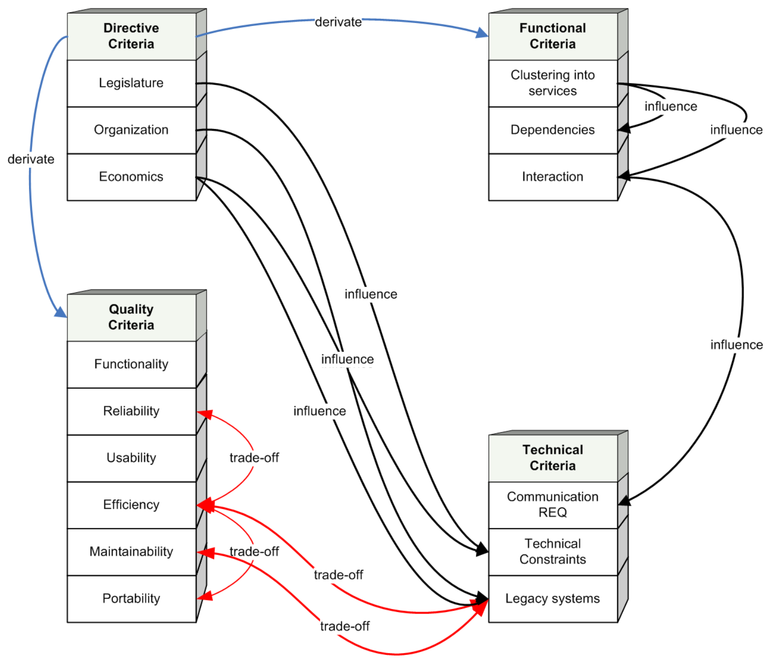


Figure 3.4: Influences between Decomposition Criteria

Within Functional and to Technical Criteria. Within the functional criteria, there is an influence from the clustering into services to the dependencies and the interaction. Information about the interaction of user services is one of the sources of origin for the communication requirements that belong to the technical criteria.

Within Quality and to Technical Criteria. In between the quality criteria there are some trade-offs, for example between reliability and efficiency as well as between portability and efficiency. If legacy systems have to be integrated, these may not have the state of the art statistics for quality characteristics in terms of efficiency maintainability, or portability. This has an influence on the statistics for the quality characteristics of the overall system, so again trade-offs have to be made.

Priority in a Trade-off. Some criteria contradict others, therefore a trade-off has to be found and agreed on, e.g., high performance versus portability.

High performance can be achieved by exploiting hardware specifics in the code. At the same time, such an hardware-specific code tailoring leads to code that can often only be deployed on exactly the hardware it has been optimized for. This contradicts the idea of high portability, i.e. to stay independent of hardware specifics to be able to easily deploy the code on a different hardware.

When both quality characteristics shall be present, a trade-off is the only solution. In case of the example of high performance versus portability, one solution is to optimize the code as far as possible without yet exploiting hardware-specific details. This includes on one hand a programming language-specific optimization, and on the other hand an optimization for a certain type of hardware units that are under consideration for deployment[2].

3.8 Impact of the Criteria on Decomposition

It is agreed upon that a system can be decomposed in different ways according to the optimization factors that are most relevant for the system development. Less obvious are the systematics of how the criteria influence the decomposition and what consequences arise from the way of decomposing a system. Major impact of the decomposition criteria can be realized in the facilitation of systems engineering and systems integration as well as in economics and return on investment.

Systems Engineering. The advantage of a functional decomposition on the engineering activities is an improved intuitive understanding of the system's functionality. The functional decomposition can still be completely independent of technical constraints. However, for distributed development, some constraints of the operational environment have to be taken into account as will be detailed in Chap. 5.

The technical decomposition, which is driven by the technical constraints and leads to a design blueprint, perfectly enables distributed development, but in practice often happens bottom-up. The better solution would instead be developing a technical architecture *after* a functional decomposition as there always has to follow a design blueprint at some stage. Therefore, the two do not contradict each other but are in fact two different stages in the development process.

Facilitation of System Integration. The functional decomposition makes integration easier as the interfaces are kept small. The technical decomposition eases integration through making technical constraints explicit. In combination with an artifact model like the one presented in Chap. 2.5.3, all relevant information is provided to allow for smooth integration.

Economics and Return on Investment. Every optimization factor that has been described in Sec. 3.2.1 is related to costs in a certain way. For example reliability, as a software that is not reliable will cause high maintenance costs after the first failed tests. Therefore, either kind of systematic decomposition and development will lead to economic benefits as long as the overhead created by the method is not too big.

Summary. This chapter introduced the criteria catalogue for system decomposition. It gave an overview of the catalogue, described the template and coherence of the criteria categories *directive*, *functional*, *quality*, and *technical*, and described each of them in detail. Furthermore, the impact of the criteria on decomposition was discussed.

[2]Think of a smallest common denominator for the hardware characteristics.

To be able to apply the listed criteria in a pragmatic way, a guiding process is needed. The information needed for that process is gathered in the description template described in Sec. 3.2.3, which was filled in for each criterion in Sec. 3.3-Sec. 3.6.

The application of the catalogue is described and shown exemplarily in Chap. 5.

Chapter 4

Subsystem Requirements Decomposition and Refinement

Contents

The decomposition criteria of the previous chapter (Chap. 3) give guidance on how to decompose a system and draw a first coarse-grained sketch of the architecture. The goal is a subsystem specification with refined requirements. In current systems development, problems often arise during system integration between the subsystems. In that case, when the subsystems do not fulfill the requirements of the system, the reason is usually missing information in the requirements specification of the subsystem. To avoid such lacking information and to guarantee compliance with the overall system requirements, it is crucial to systematically refine system requirements into subsystem requirements.

The systematic refinement of requirements is the topic of this chapter. The aim is to achieve a better understanding of how such a systematic refinement has to be accomplished for subsystem requirements to ensure satisfaction of the overall system requirements they were derived from. Furthermore, the intention is to give concrete support for realizing such a refinement.

In the following, a set of system requirements with a given system decomposition is assumed. When a decomposition has been decided on, the overall system requirements have to be deduced for the subsystems. The question is how to transform and decompose the requirements accordingly for subsystem specifications. Then, in the context of large system development with subcontractors, the subsystem specifications can be given to suppliers who implement the subsystems.

This chapter is organized as follows: In Sec. 4.1, related work for requirements refinement is discussed. In Sec. 4.2, the assumption/guarantee specifications are introduced to provide a semi-formal view of the problem. Subsequently, a subsystem model is defined (Sec. 4.3), and the distribution of a subsystem across the abstraction levels and its description are inspected.

On that basis, the actual refinement of requirements for subsystems is investigated in Sec. 4.4, detailed with case differentiation with respect to distribution of a requirement across subsystems and systematic decomposition and refinement in assumption / guarantee style (Sec. 4.5) by use of patterns (Sec. 4.6).

Finally, the decomposition and refinement of quality requirements are discussed in Sec. 4.7.

4.1 Related Work for Subsystem Requirements

Requirements refinement has been covered by research for a number of designated specification techniques, like goal modeling or behavioral specifications in labeled transition systems and state machines. Requirements decomposition has been approached through patterns with the objective of formalization.

Behavioral Specifications. A refinement approach for behavioral specifications has already been offered, for example, by von der Beeck [vdB00]. He proposes to define the semantics of the considered specification language in a compositional structured operational semantics (SOS) style with labeled transition systems (LTS). According to von der Beeck [vdB00], an adequate refinement notion should at least support reduction of non-determinism and reduction of partiality. Such an approach of formal semantics is hardly applicable for informal or semi-formal requirements.

Instead, the DeSyRe approach focuses on informal and semi-formal requirements to be widely applicable in practice.

Assumption/Guarantee Specifications. A/G specifications are used by Abadí and Lamport [AL95] for conjoining specifications in their composition theorem to prove that a lower-level specification implies a higher-level one. Henzinger et al. [HQR98] build on [AL91] and use assume-guarantee reasoning and refinement mappings on a formal system description language of reactive modules for verification purposes. They find that the success of assume-guarantee reasoning depends critically on the construction of suitable abstraction modules. In [HQRT02], the authors add an assume-guarantee rule for checking simulation.

The specification language is defined formally (the system description language of reactive modules), which is not the case for the requirements specification of this approach. However, the idea of assume-guarantee reasoning relies on plain logic, which is also applicable to statements in natural language.

Therefore, the DeSyRe approach uses that idea for requirements refinement for subsystems.

Goal Refinement. Lamsweerde [vL09] uses goal refinement to make the transition from abstract or high level goals to concrete system goals and analyze their interelation. System goals are refined by decomposing them according to their so-called *actors* by usage of patterns, for example the milestone-driven refinement pattern and the case-driven refinement pattern [vL09, p. 321ff]. Thereby, the goals are decomposed and refined until there is only one actor per goal, which turns the goal into a requirement.

In contrast, in this work, the idea is *not* to decompose the requirement to identify their actors when refining system goals, but instead to transform the system requirement into subsystem requirements for the subsystems already given by the decomposition (according to Chap. 3). The requirements that are the result of Lamsweerde's refinement serve as potential *input* for the refinement described in this work.

Requirements Patterns. The most recognized work on requirements patterns is the pattern system by Dwyer et al. [DAC99] as these patterns have been found in an empirical evaluation of over 550 specifications. The patterns are classified according to the system behaviors they describe in terms of occurence and order of actions. Smith et al. [SACO02] used these patterns in a "disciplined natural language representation" (by means of templates with lists of alternative phrases) to specify commonly-occuring properties with the aim of eliciting "precise and rigorous requirements from people who are unlikely to be fluent in temporal logics or other specification formalisms" [SACO02, p. 12].

The four basic patterns are: *Response*, *Precedence*, *Existence*, and *Absence* [SACO02, p. 14]. Any other pattern can be constructed by using composition. According to Zave and Jackson [ZJ97], there are two *expressive capabilities* needed. The first one is that a sufficiently expressive requirements language must provide for the declaration of a finite collection of *action types* that indicate whether an action is environment-controlled or machine-controlled and whether an action is shared or unshared [ZJ97, p. 10]. As requirements do not describe the solution itself but the environment under influence of the machine, the case of an unshared, machine-controlled action does not occur. The second necessary expressive capability is the *full ability to state constraints on actions* in all categories. Assertions can be stated about all actions either indicative (about the environment) or optative (as requirement) by using safety and liveness properties [ZJ97, p. 11]. According to Alpern and Schneider [AS87], any property in linear-time temporal logic can be expressed as combination of safety and liveliness properties. These expressive capabilities are also covered by the requirements patterns.

The requirements patterns in [SACO02] are characterized as follows.

- Response: A specified stimulus is followed by a response. Example: "If a discard-signal is received, the current entry is deleted." [Ris07, p.37]
- Precedence: A specified action cannot occur until it has been preceded by an enabling event. Example: "The code shall be sent only after a code was sent from the queue." [Ris07, p. 37]

- Existence: An action must occur in the system execution. Example: "All radio signals are added to the queue." [Ris07, p.31]
- Absence: An action must not occur in the system execution. Example: "The component may not produce short circuits." [Ris07, p.44]

These patterns are helpful to understand the content of the requirements and proceed in the direction of formalization, but they are not helpful for decomposing and refining the requirements for subsystems as their objective is *not* to provide a decomposable structure. Instead, as presented in this work, different cases with respect to decomposition can apply (Sec. 4.5) and work for all patterns in [SACO02], thereby ensuring general applicability of the approach and confirming the authors' findings about the general distribution of occurrences of their requirements patterns.

Further work on requirements patterns is presented by Fleischmann [Fle08]. The idea is to formalize initial informal requirements issued by stakeholders in order to enhance their quality and ease transition to design. This is achieved by classifying the requirements and filter the functional usage requirements, then decompose and model the single contained natural language elements. Subsequently, the requirements are ordered in a usage service hierarchy, which significantly increases the amount of requirements due to requested completeness.

The aim of the work is to provide a transition from natural language requirements to formalized usage service descriptions. Instead, DeSyRe decomposes requirements according to an initial design decomposition for separate subsystem requirements specifications that can be passed on to subcontractors for realization of the subsystems.

Identified Shortcoming in Literature. As described in the preceding paragraphs, there is guidance on the refinement of some specification techniques, i.e. formal behavioral specification and goal modeling. However, there is no systematic approach to the decomposition and refinement from system requirements to subsystem requirements for a given system decomposition. The work at hand addresses the identified shortcoming by use of requirements patterns and assumption/guarantee specifications.

4.2 Prerequisites for Requirements Refinement

On the basis of a given logical system decomposition, the goal is to develop a subsystem specification with refined requirements, as depicted in Fig. 4.1. On the system level (denoted by C in Fig. 4.1), the requirements are black box (top of left hand side), but the given initial decomposition (right hand side of Fig. 4.1) makes the first step to design. Then, one subsystem (A and B in Fig. 4.1) of the first white box specification is used to describe the requirements in a black box manner on the subsystem level (bottom of left hand side).

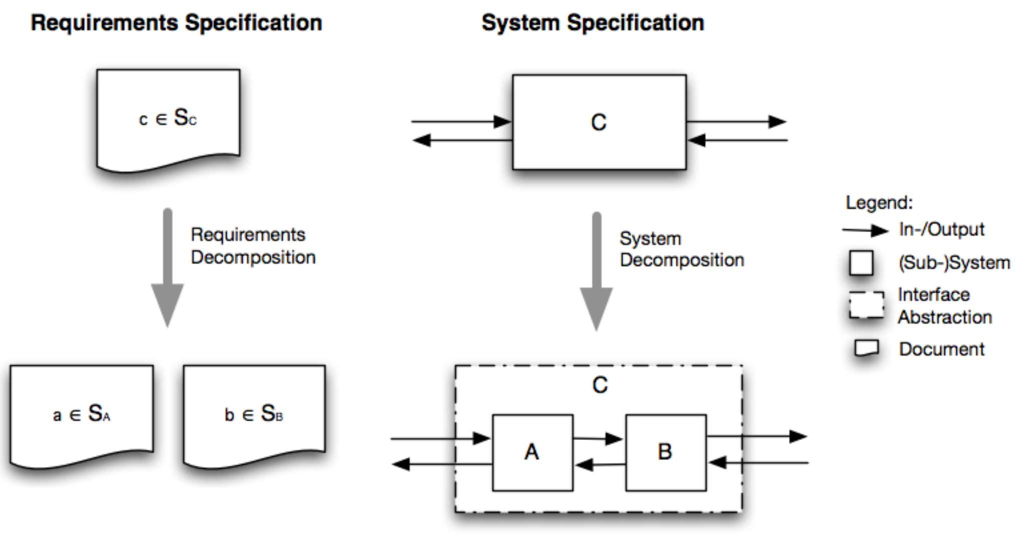


Figure 4.1: Transition from System Requirements to Subsystem Requirements.

4.2.1 Assumption / Guarantee Specifications

When decomposing and refining a requirement, it is necessary to ensure that the resulting requirements for subsystems do indeed guarantee the overall system requirement. System components are designed for environments that satisfy certain assumptions. If those assumptions are fulfilled, the components commit themselves to certain guarantees. The same idea can be applied to requirements by using assumption/guarantee (A/G) specifications with logical implications.

The first formal method based on A/G specifications that received wide attention was the pre/postcondition style of Floyd/Hoare logic [Hoa69], which provided the foundation for further methods.

Assumption/guarantee specifications consist of two parts: an assumption and a guarantee. The assumption describes properties of the environment in which the specified component is supposed to run. The guarantee describes what the component has to fulfill in case the assumption is satisfied by the environment [BS01, Chap. 12]. Denoted as formula in Eq. 4.1, the assumption implies the guarantee.[1]

$$A(input, output) \Rightarrow G(input, output) \tag{4.1}$$

Specification with A/G has also been used in [Bro05] for describing services. The idea of A/G reasoning relies on plain logic, which is also applicable to statements in natural language as often used for requirements specifications. In the following, this idea is used for the capturing and the refinement of requirements. Approaching requirements by using A/G has also been recognized as issue for future research by [Bro10].

[1]Although the approach is intended for deterministic systems, the assumption restricts the input and at the same time references the past output [Bro95]. The reason is that requirements do usually not compose a total (in the sense of complete) specification but describe an underspecified system, which can be interpreted as non-determinism.

Refining the requirements specification by means of assumptions and guarantees is chosen for the approach described in this work because exactly and exclusively the *relevant* constraints from the surrounding system environment are captured by specifying the requirements using A/G. A specification technique itself can not guarantee for making appropriate use of it, therefore the precondition is that the requirements engineer puts effort into an appropriate description of the overall system requirements with A/G and they are elaborated sensibly.

The challenge when refining the system requirements for a subsystem is to ensure that all relevant information is present but no overhead information is dragged along. Therefore, when a requirement is represented as guarantee and the constraints under which the requirement needs to hold are represented as assumption, *exactly* the constraints which are *relevant* to fulfill the requirements are present within the assumptions. Consequently, a requirements specification in A/G style allows to extract the relevant requirements for a specific subsystem.

4.2.2 Semi-formal View of the Problem

The motivation for the decomposition of the system into subsystems is the assignment of subsystems to subcontractors for distributed development. The subcontractor has to be provided with an adequate subsystem requirements specification, hence the question to answer is: *"For a given system requirement, how to develop the requirements for the identified subsystems?"*

The given input is therefore:

- A system $\mathcal{C}$ and an initial decomposition into the subsystems $\mathcal{A}$ and $\mathcal{B}$.
- A set of requirements for the system, captured in a system requirements specification S_C.

The envisioned result is:

- Sets of refined requirements per subsystem, the subsystem requirements specifications S_A and S_B.
- A mapping of these refined requirements so that their composition implies the overall system requirements, so that the (re-)combination of S_A and S_B implies S_C.

To describe how to transform and decompose the requirements accordingly, it is helpful to use a formal view onto system decomposition as depicted in Fig. 4.2. In the figure, the overall system $\mathcal{C}$ is decomposed into subsystems $\mathcal{A}$ and $\mathcal{B}$, x_1 and x_2 are the input and y_1 and y_2 are the respective output. The subsystems are related via local channels z_1 and z_2, with "channel" meaning typed communication channels. The simple model in Fig. 4.2 encompasses all possible decompositions, as a communication channel may or may not be used in a specific system specification.

In order to formalize the transition from requirements for the overall system and requirements to the subsystems, the following questions need to be answered:

1. What is an adequate subsystem model to analyze the requirements refinement?

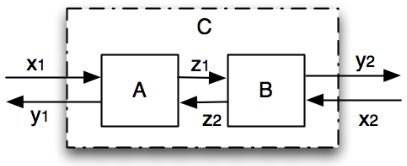


Figure 4.2: Formalized System Decomposition.

2. How to decompose the requirements for $\mathcal{C}$ into requirements for $\mathcal{A}$ and $\mathcal{B}$?

Question number one is answered in Sec. 4.3 and question number two in Sec. 4.5. Within the answer to the second question, it is also interesting if all types of requirements can be decomposed and for which types of requirements a refinement is compositional with regard to the system decomposition. Thereby, compositionality means that a requirement can be decomposed and then composed again with the result fulfilling the original requirement. In explicit, decomposition is considered a reversible action. Refinement and decomposition are clearly distinguished from each other: refinement is the act of adding details to a model, decomposition is the act of partitioning a model. However, when specifying the subsystem requirements with a set of system requirements at hand, often both is performed conjointly in one step.

The following section presents the subsystem model that serves as basis for analyzing the requirements refinement.

4.3 Subsystem Modeling

In the subsequent sections, a model for subsystems based on the system model introduced in Sec. 2.3 is presented. As stated in Sec. 1, the standard IEEE definition of the terms *system*(Def. 1.1) and *subsystem*(Def. 1.2) [JM90] is used.

The scope of a subsystem depends on the regarded abstraction level. A subsystem is again modeled and documented like a system, as the differentiation between *system* and *subsystem* depends on the scope or context in which a system is considered[2]. The context of a subsystem is derived from a part of the context of the complete system plus new information that originates from the surrounding architecture.

Moreover, all information about the boundaries of the subsystem has to be documented explicitly to allow to quickly get an overview of all relevant constraints imposed by the context. To improve reusability, the context boundaries have to be described such that the required conditions can thereby be reproduced in another system. This means that the right degree of abstraction has to be chosen for the formulations. With respect to the content, the subsystem boundaries include the interface specification and constraints imposed by both the operational environment and the project settings (or business context).

For example, the following requirement does not choose the right degree of abstraction: “The controller requires the input speed sensor SpeedySens

[2]In other words: “A system is a subsystem is a system.”

xyz2000". Here, *SpeedySens* stands for the brand and the *xyz2000* for the special model that has been found applicable. This requirement specification does not tell which exact characteristic of the input speed sensor is required. Therefore, the description "The controller requires an input speed sensor with trouble codes according to the 2008 standard No. 4711 (cf. Ref. 4711)" is more helpful and the information "SpeedySens xyz2000 was used successfully during pre-serial prototyping." should instead be added in the corresponding context artifact on the hardware / software level.

A subsystem is defined with different scope on each abstraction level, as will be detailed in the following section: In terms of a feature on the usage level, as logical component on the logical architecture level and as software or hardware component on the technical architecture level. For the approach at hand, the emphasis lies on the usage level and the logical architecture level. The subsequent section presents an adoption of the architecture model introduced in Sec. 2.3 for the description of subsystems.

4.3.1 Definition of a Subsystem Model

As stated above, a subsystem is modeled and documented like a system, therefore the subsystem model uses the abstraction levels of the system model from Sec. 2.3 by Broy et al. [BSW^{+}08], namely usage services, logical architecture and technical architecture. The reason to detail a specification over three abstraction levels is that each abstraction level focusses on certain aspects and, while moving from the most abstract level to the more concrete ones, each level adds the examination of one particular problem. The usage service level abstracts from the internal structure and behavior of the system, and the logical architecture level abstracts from the distinction of hardware and software. While the technical architecture level is still depicted in Fig. 4.3, this work focusses on the upper two abstraction levels and their relation.

Usage Services. For easier discussion, the usage services of the subsystem depicted in Fig. 4.3 are labeled with capital letters and the logical components are labeled with numbers. The usage services are hierarchically structured, where S is the main usage service (Def. 2.1) of the system that is decomposed into subservices A and B, which are in turn decomposed into the subservices C, D, E, F, and G. The hierarchical decomposition leads to a tree-like shape, with relations between the usage services depicted as directional or bidirectional dependencies. Thereby, relations can be transitioned and refined such that they occur only on the leaf level of the tree (cf. the thesis of Rittmann on modeling usage behavior [Rit08b]). Due to the hierarchical decomposition with refinement, the leaf level of the service tree covers the complete interface of the usage service S and its subservices [Rit08b]. The dashed rectangle in Fig. 4.3 illustrates an abstraction of the usage services reduced to their interface. Examples for usage service graphs can be found in the case study in Fig. 6.2 and Fig. 6.5.

Logical Architecture. On the logical architecture level, the components that realize the subsystem are denoted with the numbers 1 through 6. As the logical components are interconnected, not every component necessarily has

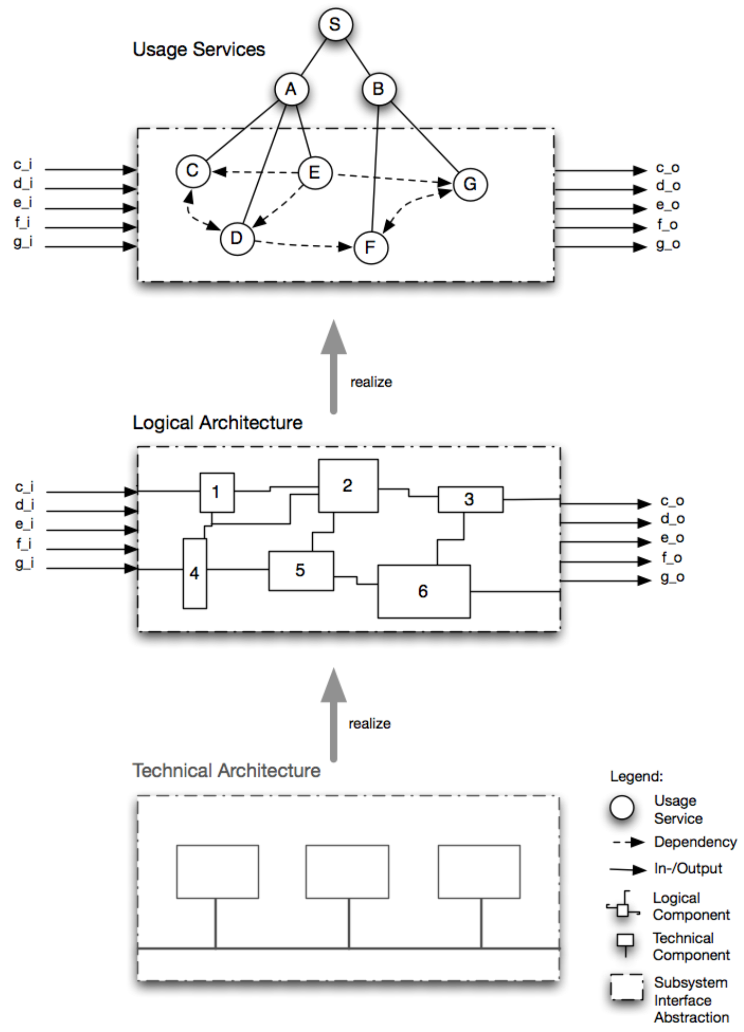


Figure 4.3: Subsystem Model.

an interface that is visible at the border of the logical subsystem. The interface is illustrated by the inputs c_i through g_i and the outputs c_o through g_o already known from the usage service level. Again, the dashed rectangle illustrates the abstraction of the logical architecture reduced to the interface.

Technical Architecture. The technical architecture level describes the realization composed by hardware and software components and the deployment (as introduced in Sec. 2.3). This lowest level of abstraction is depicted here for completeness of the subsystem model, but is not used further during the discussion of requirements decomposition.

Interface Refinement. The interface of the logical architecture of a system is a refinement of the interface of the respective usage service hierarchy [Bro07]. However, for the systematic decomposition and refinement of requirements, a discussion about interface refinement does not bring any new insights. Therefore, a possible interface refinement is not discussed any further in the following.

Capture in Assumptions and Guarantees. All relevant information with respect to the system environment of the subsystem, i.e., information about the relations with the surrounding system, can be captured by using an assumption / guarantee style specification as introduced in Sec. 4.2.1. Thereby, the assumption denotes the information about the outside context, and the guarantee denotes what the system is required to deliver given that the surrounding environment fulfills the assumption.

The following section discusses the distribution of a subsystem across the different abstraction levels and their relation.

4.3.2 Subsystem Distribution across Abstraction Levels

Within this section, the identification of corresponding modeling elements on the different abstraction levels depicted in Fig. 4.3 is discussed. This means reasoning about how to determine the corresponding logical component (logical architecture level) responsible for a certain usage function (usage level) in top-down direction *and* how to determine the corresponding usage function realized by a certain logical component in bottom-up direction.

Subsystem Units on Each Level. The usage service level units (cf. Fig. 4.3), called services, represent chunks of user-perceivable functionality. The logical level units, called logical components, represent entities that interact as logical items to realize the services described on the usage level. The technical level units, called software and hardware components, represent the controllers, the input and output devices, and the communication channels to realize the functionality described through the logical level.

The implementation-relation between subsystems on the different abstraction levels is not necessarily 1:1. In general, the relationship between units of one level to units of another level is many-to-many (n:m). As illustration, Fig. 4.4 depicts the system slice for one logical subsystem. This is due to the different kinds of structural units that are used to decompose a system on a specific level of abstraction. It is, for example, rarely the case that a specific usage function can be realized by one logical component, furthermore not all logical components are visible at the system interface.

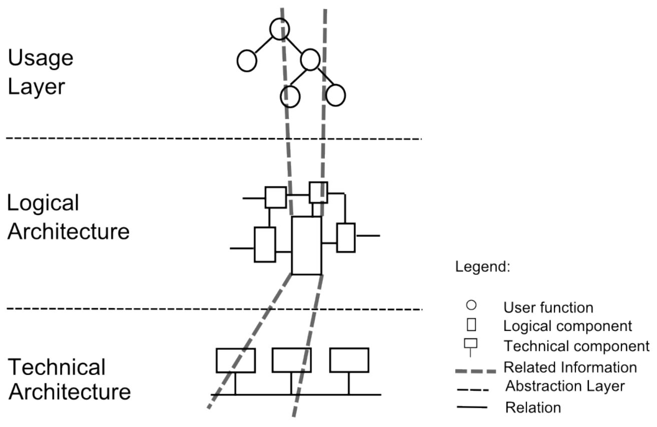


Figure 4.4: System Slice based on a Logical Subsystem.

Transition from Usage Level to Logical Level. In the transition from the usage service level to the logical level (cf. Fig. 4.3), a system unit usually implements a number of services (at least in parts). Each of them is represented by one logical unit on the logical level. The difficulty is that not every system entity in that sense is already identified on the usage level. This means that there will be logical components on the logical architecture level that did not exist as explicit actors in the service descriptions on the usage level, because they are used for reasons of distributed development.

For example, the service descriptions of the radio frequency warning system explain that a warning is shown on the display and that a warning tone is created in certain circumstances, but the service description does not say that there is a controller that calculates whether a warning has to be generated at all. Consequently, the logical components “display” and “tone system” can be derived from the service description but not the “controller”. The reason for this is that the display and the tone system are visible at the system interface, while the controller is not user-perceivable. Adding the latter as logical component is part of the engineer’s effort that has to be performed by a human developer as genuine design work.

In order to trace the distribution of a logical component bottom-up to find the related usage functions, part of the service descriptions can be identified via the entities that are present as roles in service descriptions and correspond to specific logical components. Another part of the necessary knowledge is captured in the system context and can hardly be sliced according to the subsystem as that context is equally relevant and mandatory for all subsystems.

Tracing the distribution of a described user service of the usage level top-down to find all related logical components would be even more complicated if the goal was not to end up with nearly the complete realization on the lower abstraction levels. Instead, a group of closely related services might be chosen with use of engineer's expertise. Following the decomposition of the system as described in Sec. 5.4 this will lead to one or at least very few logical components for one group of closely related services, so one can speak of the extraction of a logical subsystem in this case as well.

Transition from Logical Level to Technical Level. During the transition from the logical level to the technical level (cf. Fig. 4.3), each of the logical components gets assigned a technical component that realizes the specified behavior. This technical component can be a hardware component, or a software component, or a combination thereof. Such a 1:1 mapping is a canonical way to perform the deployment. Unfortunately, this is not reasonable as, due to cost efficiency, hardware components may be used by different software components and different software components can be deployed on the same hardware component. For example, the radio frequency warner has a controller with relatively few lines of code that will not require an extra electronic control unit (ECU) in the car. Instead, the software will be deployed on an ECU where there is still capacity available.

To trace the distribution of a logical component across the abstraction levels, all technical components related to that same logical component have to be traced and included. For the case where there are no technical components related to more than one logical component at a time, this is manageable. But in the case of shared technical components, this can lead to a high number of components that have to be included in the subsystem specification.

When looking at the distribution of a described technical component bottom-up across the logical architecture, the same difficulty of multiple logical components belonging to different multiple technical components arises. There can be logical components that have nothing to do with each other in terms of contributing to the same functionality but that are deployed on the same hardware component. For that reason it is not expedient to extract a subsystem specification from the technical level bottom-up across the abstraction levels.

Due to the discussed distribution of a subsystem across the abstraction levels, the extraction of a subsystem specification or slice of a system as depicted in Fig. 4.4 will therefore in most cases be based on a logical subsystem.

In addition to the examples given above, further illustrations of what subsystems are on the different abstraction levels and how they relate to each other or how their distribution differs on the subsequent abstraction levels are given in the case study in Sec. 6.1.

4.3.3 Subsystem Description across Abstraction Levels

Describing a subsystem across the abstraction levels of the subsystem model (cf. Fig. 4.3) requires a model for the slice of a system and a point of origin for the slicing. The starting point has been chosen as a logical subsystem on the intermediate abstraction level as illustrated in Fig. 4.4 for the reasons discussed in Sec. 4.3.2.

A logical component represents a unit that encompasses an interface, structural and behavioral description [BSW$^+$08]. Formally, a logical component possesses a syntactic and a semantic interface. The syntactic interface defines how the logical components can be connected to each other by their inputs and outputs. The semantic interface specifies the behavior of a logical component. A user services on the usage level may have partial behavior, i.e., a user service may be defined only for expected inputs. Thereby, the legal inputs are oftentimes only a subset of the syntactically possible inputs. In contrast, a logical component always possesses a total behavior, which means it delivers a well-defined result for each syntactically possible input at each point in time.

Tracing to Services. The respective extract of the system description on the usage level is a collection of service descriptions, depicted as service hierarchy in Fig. 4.4. The term *service* (Def. 2.1) is used as defined by Broy [Bro05] and used by Rittmann [Rit08b] for the modeling of usage[3].

The associated service descriptions can be traced by identifying the respective users (human or other system) in the service descriptions and relate them to the logical components of the subsystem of interest. Subsequently, all of the service descriptions are collected that feature the respective users. As indicated in Fig. 4.4, this will lead to services being included that are not exclusively realized by the logical components of the subsystem of interest but only share some of the actors. This means, that tracing the service descriptions for all logical subsystems will *not* lead to a disjunct partition. Instead, there is a many-to-many mapping between services and logical components.

At the same time, not all logical components will coercively have a corresponding, explicitly visible user within the service descriptions on the usage level. This "gap" is due to the earlier mentioned engineer's effort when developing a logical architecture, because not all logical components are visible at the system interface. Therefore, a system slice will always lead to a number of relevant service descriptions for a logical component.

Tracing to Hardware/Software. The respective extract of the system description (the specification of the system slice) on the technical level is a collection of software component descriptions, hardware component descriptions, and an adequately reduced deployment diagram. In general, there is a many-to-many relation between logical and technical components.

The descriptions are collected by identifying the corresponding technical components, whether they be hardware, software, or combination thereof, for the logical components of the subsystem of interest. These technical components may not solely belong to the subsystem of interest but instead be shared by multiple subsystems. Again, similar to the usage level, tracing the technical components for all logical subsystems will not lead to a disjunct segmentation.

4.4 Refinement Application Guideline

The following brief application guideline sums up the refinement of requirements for subsystems in a process overview before detailing the ideas and reasoning for the process steps in Sec. 4.5.

[3]"A service is a piece of functionality - abstracting from technical structure. It is described by a black box view relating inputs to outputs and hiding the internal realization. It is a partial behavior, i.e. it might not be defined for all possible inputs in each situation." [Rit08b, p. 175]

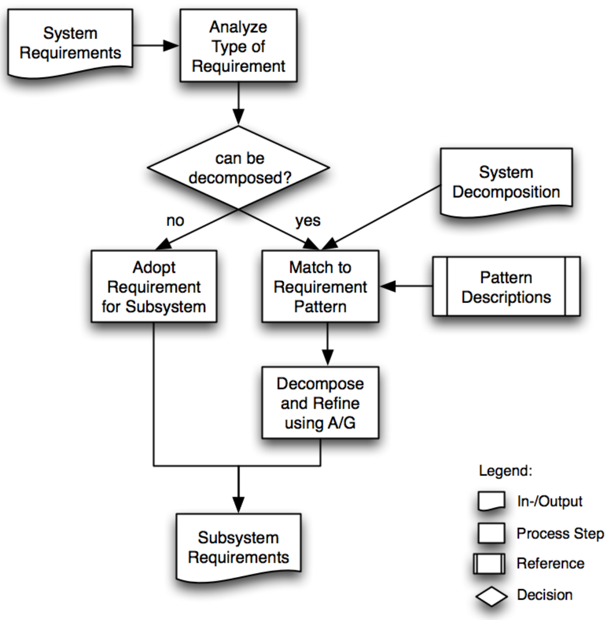


Figure 4.5: Decomposition and Refinement Process Description.

The steps to take for refining a system requirements specification are depicted in Fig. 4.5. The process input is the system requirements documentation. As a certain aspect of the system realization is anticipated by the decomposition into subsystems, the transition from system requirements to subsystem requirements usually includes both a decomposition and a refinement of the system requirements. For each system requirement, the following steps are performed:

- *System Requirements:* The input (depicted top left in Fig. 4.5) for the depicted steps are the system requirements as A/G specification (introduced in Sec. 4.2.1).
- *Analyze Type of Requirement:* The requirements are assumed to be detailed enough that it is possible to differentiate to which subsystem they belong. This assumption can be made for two reasons: First, the system decomposition is already decided, which can only be the case if the requirements were detailed enough to identify the subsystems. Second, if this does not apply for all requirements at this stage of the development process, the requirements engineer has to refine these requirements before continuing the process.[4]

  The following steps are performed for every system requirement. In case only one subsystem is of interest, the requirements engineer performs the steps only for the system requirements that contain a reference to the respective subsystem (e.g. the subsystem realizes one step in a functional

[4]If the requirement is a rather abstract goal and can not be operationalized, the compliance of the system has to be assured by defining acceptance criteria and/or tests. [Gli07]

description) in either their assumption or their guarantee. The latter applies to all functional requirements. In case of quality requirements, the decision, whether a requirement is relevant for a specific subsystem, requires the expertise of the requirements engineer.

- *Can be Decomposed:* In this step, the requirements engineer decides whether a requirement can be decomposed at all or has to be adopted (top of Fig. 4.5). If the requirement refers to only one subsystem, it is adopted. If it refers to several subsystems, it has to be decomposed. Decomposition requires compositionality, which can be guaranteed in case of a functional system specification [Bro95].[5] For quality requirements, compositionality is still under active research, e.g. for security [Can01, PM05], and often requires additional properties for composition, e.g. a model for composing probabilities. Therefore the applicability for quality requirements is not guaranteed in general, but further analyzed in Sec. 4.7. Still, the approach was applied successfully to a number of examples during a case study.

"No": *Adopt Requirement for Subsystem:* In case of requirements that apply to only one subsystem, adopt the system requirement for the subsystem specification as no refinement is possible. Straight-forward adoption of a requirement causes redundancy, which is usually not wanted within requirements documentation. However, in this case the redundancy is expedient as the subcontractor does not receive the whole system specification but only the subsystem specification with just enough information to meet his needs.

"Yes": *Match to Requirement Patterns:* There are three patterns according to which a requirement can be decomposed, two special cases that apply to most requirements, and a general one than applies otherwise. In this step, the requirements engineer decides which pattern is applicable for the requirement at hand with the aid of the pattern descriptions.

- *Decompose and Refine using A/G:* The final step to deduce the subsystem requirement is to decompose and refine the system requirement with A/G specification according to the identified pattern.
- *Subsystem Requirements:* After the requirements engineer has performed the described steps for all requirements, the output is a complete set of subsystem requirements specifications.

The process steps of decomposition in combination with using assumptions and guarantees of the subsystem requirements imply compliance of the overall system requirement. Therefore, an appropriate refinement covers and implies compliance of the system requirement. Moreover, the advantage of the whole system specification being written in A/G style is the capability of capturing exactly the *relevant* information (see Sec. 4.2.1).

The concepts for the process steps and the actual analysis and refinement are detailed in the following Sec. 4.5.

[5]Strong causality is required for ensuring that the performed decomposition is correct.

4.5 Case Differentiation for Requirements Distribution

As stated in Sec. 4.2.1, by using assumption/guarantee specifications, exactly all relevant information with respect to the system is captured. S_C (as introduced in Sec. 4.2.2) is a set of requirements, also called the system requirements specification of the usage service level (see Fig. 4.3). The following cases have to be differentiated when decomposing and refining single requirements:

1. A one-to-one transition of a system requirement $c \in S_C$ to a subsystem requirement $a \in S_A$ is performed, where the requirement is plainly adopted. In that case, the subsystem shows the complete functionality required by the system requirement (detailed in Sec. 4.5.1).
2. A one-to-many refinement of a system requirement to requirements for a set of subsystems is performed, so that S_A combined with S_B implies S_C is guaranteed (detailed in Sec. 4.5.2).[6]

These cases encompass all possibilities to decompose and refine a system requirement into subsystem requirements — either there is one match with a responsible subsystem, many matches, or a match with all subsystems.

For each system requirement of a given system specification, one of the above cases applies. The following sections give the case differentiation, detail the refinement and illustrate each of them with examples.

4.5.1 One-to-one Transition of Requirements

If a specific subsystem $\mathcal{A}$ provides the complete functionality or property that a specific system requirement $c \in S_C$ demands, a refinement might not be necessary. Instead a straight-forward adoption of the system requirement c into the subsystem specification S_A occurs.

An example for the direct adoption of an overall system requirement stems from the driver assistance systems: "The system automatically holds the distance to the preceding vehicle."

In this case, there is no refinement necessary, the requirement can be adopted directly for the responsible subsystem "Adaptive Cruise Control". This is not a frequent case because requirements will usually be phrased according to the degree of detail on the respective abstraction level of the system specification. The one-to-one mapping will only occur in cases where one subsystem exhibits the complete functionality demanded by the requirement.

For a detailed specification of the adaptive cruise control, it is nevertheless necessary to do a refinement later on. For that refinement, the same rules as for the one-to-many transition described in Sec. 4.5.2 apply.

4.5.2 One-to-many Transition of Requirements

In the case of a one-to-many transition, a specific system requirement $c \in S_C$ of the usage service level (cf. Fig. 4.3) is realized by more than one subsystem, let these subsystems be $\mathcal{A}$ and $\mathcal{B}$ (as in Fig. 4.1).

[6]The one-to-all case is a subset of the one-to-many case and therefore its analysis will not bring further insights, which is why it is not discussed any further in the following.

A specific system requirement c shall be decomposed into subsystem requirements $a \in S_A$ and $b \in S_B$ such that, in the case of compliance of the subsystem requirements, the compliance of the overall system requirement is ensured.

$$\bigwedge_{a \in S_A} a \wedge \bigwedge_{b \in S_B} b \Rightarrow \bigwedge_{c \in S_C} c \tag{4.2}$$

For a valid refinement of all system requirements $c \in S_C$ of a system $\mathcal{C}$ within the subsystem requirements $a \in S_A$ and $b \in S_B$ of the subsystems $\mathcal{A}$ and $\mathcal{B}$, the conclusion is that if all subsystem requirements are met for both $\mathcal{A}$ and $\mathcal{B}$, all system requirements for $\mathcal{C}$ are fulfilled.

This assurance is guaranteed by the use of assumptions and guarantees, where, in explicit, "the assumption describes exactly those aspects of the environment that are significant for the functioning of the specified component." [BS01, p. 213] and in case of compliance to that assumption the component gives a certain guarantee about its behavior.

In the case of decomposition, each subsystem makes a certain assumption about its environment and if that assumption is fulfilled, the subsystem can give a certain guarantee. The sum of the guarantees implies the compliance of the overall system requirement. As stated at the beginning of the chapter, it is assumed that there is a given system decomposition for which adequate subsystem requirements are desired.

It is sufficient to analyze the decomposition patterns of a system with two subsystems. The cases with more subsystems can be reduced to that basic case by introducing intermediate steps, where the system is decomposed into two subsystems and then, iteratively, each of the subsystems is decomposed again until the desired decomposition is simulated.

4.6 Decomposition and Refinement Patterns

This section describes the three decomposition patterns and provides illustrating examples from case studies. In the following, assumptions will be denoted as predicates A_X with the index indicating the respective (sub-)system specification the assumption is taken for and guarantees will be denoted as predicates G_X with the index indicating the respective (sub-)system specification the guarantee is given by.

Requirement vs. Service. In the requirements documentation, the description of a specific usage service is usually split up into more than one requirement. A usage service is often complex and it does not make sense to try to decompose a complex service description as it complicates the specification of the subsystems instead of easing the deduction. A complex service is rather described in a list of requirements.

Consequently, the general case where input and output are both shared by two subsystems rarely occurs for the task of decomposition of a set of requirements. Instead, for the majority of requirements, there are two special cases that form subsets of the general case.

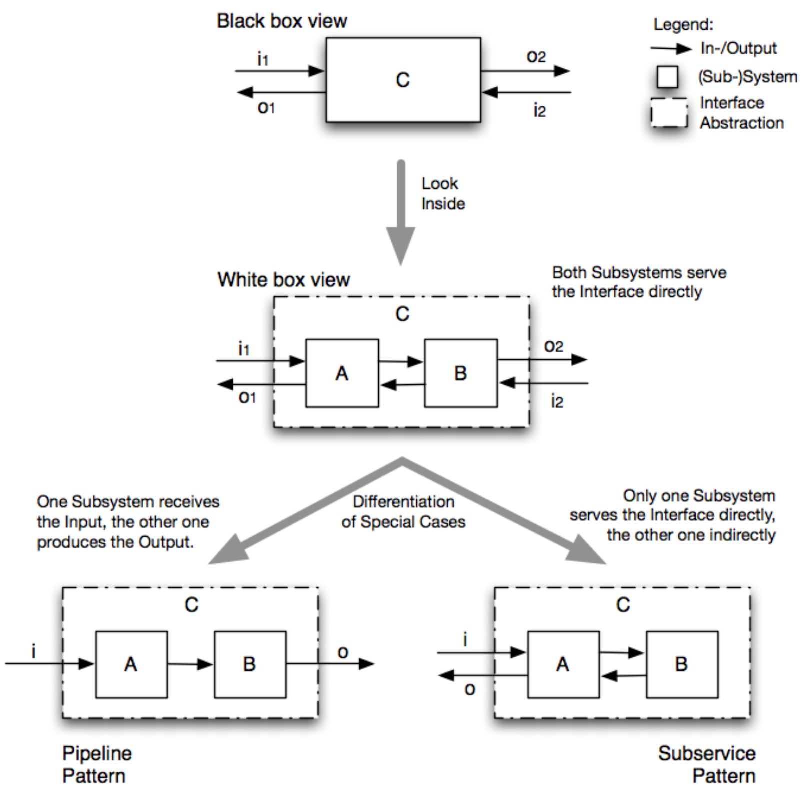

Black box view
i1
o1
C
o2
i2
Legend:
In-/Output
(Sub-)System
Interface Abstraction
Look Inside
White box view
Both Subsystems serve the Interface directly
C
A
B
i1
o1
o2
i2
One Subsystem receives the Input, the other one produces the Output.
Differentiation of Special Cases
Only one Subsystem serves the Interface directly, the other one indirectly
C
i
A
B
o
C
i
o
A
B
Pipeline Pattern
Subservice Pattern

Figure 4.6: Case Differentiation for Decomposition Patterns.

Special Cases. For two subsystems, there are two special cases that can be applied for many requirements, depicted in Fig. 4.6. As stated in the preceding paragraph, the general case where input and output are both shared by two subsystems (center of Fig. 4.6) is usually not captured within a single requirement. For reasons of completeness it is necessary to discuss the general case here, but for most requirements, a subset of (easier) special cases is sufficient to perform the decomposition.

A single requirement can in many cases be decomposed by using one of the two decomposition patterns *subservice* (bottom right side of Fig. 4.6) and *pipeline* (bottom left side of Fig. 4.6). These terms have been chosen because they resemble ideas from the architectural patterns of "main program and subroutines" (here called subservice) and "pipeline" that are described, inter alia, by Vlissides et al. [VCK96].

For single requirements, the two decomposition patterns do already cover most cases as the differentiation is made only between a decomposition where the input and the output are produced by different subsystems (e.g. i_1 and o_2 in Fig. 4.6) and a decomposition where input and output are produced by the same subsystem with the subsystem using a second subsystem as service (e.g. i_1 and o_1 in Fig. 4.6).

For the requirements that do not apply to either of the special cases, the general pattern is used. Decomposition into more than two subsystems is based on a composition of these three patterns. In the following, first the pipeline pattern is explained, because it is the most simple case, then the subservice pattern, and finally, the general pattern.

4.6.1 Pipeline Decomposition Pattern

The pipeline pattern is used in the case when both subsystems provide part of the interface for fulfilling the requirement. Before explaining the pattern in detail, it is illustrated with an example.

Example. To illustrate the pipeline decomposition pattern, an example requirement from the radio frequency warning system is used. An overall system requirement on the usage service level (Fig. 4.3) is:

> "In case of speeding, the driver shall be warned by the system." [Ris07]

The interface is described by an input channel *speed info* and an output channel *display info*. The system receives the signal from the wheel sensors periodically and detects whether the driver is speeding. In case the vehicle is faster than the allowed speed stored by the system, the driver is warned by use of the display. The assumption and guarantee[7] on the system level are:

- $A_{RFW}(\textit{speed info})$:
  The current speed information is available.
- $G_{RFW}(\textit{speed info, display info})$:
  The system displays a warning to the driver if the speed is too high.

The relevant subsystems on the logical architecture level (Fig. 4.3) are the RFW Control and the Display Control, depicted in Fig. 4.7.

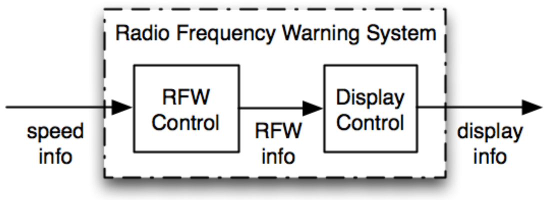


Figure 4.7: Pipeline Decomposition of an Example of the RFW System.

Compliance to the overall requirement by the subsystem requirements is assured by the following assumptions and guarantees:

- $A_{RFWControl}(\textit{speed info})$:
  The current speed information is available.

[7]The introduced scheme of A/G from Sec. 4.2.1 provides assertions on data streams while some of the following assertions are phrased as actions. The difference is only a nuance, and no further consequences introduced by that difference could be detected.

- $G_{RFWControl}(\textit{speed info, RFW info})$:
  The system sends information about legality of the current speed.
- $A_{DisplayControl}(\textit{RFW info})$:
  The information about legality of the current speed is available.
- $G_{DisplayControl}(\textit{RFW info, display info})$:
  The system displays a warning to the driver in case of speeding.

If both guarantees hold, the requirement is satisfied. As one guarantee is the assumption for the next subsystem, respectively, the satisfaction of the requirement does not depend on further external influences.

The pipeline decomposition example makes assumptions about input *speed info* and gives a guarantee for *display info*. Thereby, *RFWControl* uses *DisplayControl* and forwards to *DisplayControl*, so that *DisplayControl* triggers *display info*. The following implications apply by using plain logic:

- $A_{RFWControl}(\textit{speed info})$: The current speed information is available.
- $A_{RFWControl}(\textit{speed info}) \Rightarrow G_{RFWControl}(\textit{speed info, RFW info})$:
  The system sends information about legality of the current speed.
- $G_{RFWControl}(\textit{speed info, RFW info}) \Rightarrow A_{DisplayControl}(\textit{RFW info})$.
- $A_{DisplayControl}(\textit{RFW info}) \Rightarrow G_{DisplayControl}(\textit{RFW info, display info})$:
  The system displays a warning to the driver in case of speeding.
- $G_{DisplayControl}(\textit{RFW info,display info}) \Rightarrow G_{RFW}(\textit{speed info,display info})$
  as defined at the beginning of the example description.
- This complies with the original system requirement
  "In case of speeding, the driver shall be warned by the system." $\square$

Hence, the example decomposition for the pipeline pattern works. Subsequently, the universal description of the pattern follows.

General Description Pipeline Pattern. In case of the pipeline decomposition pattern, both subsystems provide part of the interface for fulfilling the requirement, as depicted in Fig. 4.8.

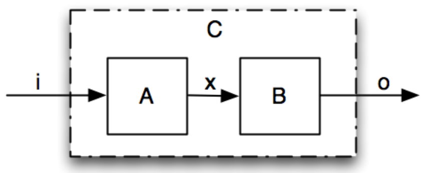


Figure 4.8: Pipeline Decomposition Pattern.

For the overall system $\mathcal{C}$ with communication channels i and o (Fig. 4.8), the guarantee G_C with respect to output o relies on assumption A_C for input i.

$$A_C(i, o) \Rightarrow G_C(i, o) \tag{4.3}$$

The *pipeline* is formed by the subsystems $\mathcal{A}$ and $\mathcal{B}$. Subsystem $\mathcal{A}$ receives input i, produces output x, which again is taken as input from subsystem $\mathcal{B}$ to produce output o. For the pipeline pattern, the decomposition results in the following:

- Subsystem $\mathcal{A}$: $A_A(i,x) \Rightarrow G_A(i,x)$
- Subsystem $\mathcal{B}$: $A_B(x,o) \Rightarrow G_B(x,o)$
- Furthermore, subsystem $\mathcal{A}$ satisfies the assumption of subsystem $\mathcal{B}$.

Hence, the composition is a behavioral refinement[8] of the original specification:

$$\big(A_A(i,x) \Rightarrow G_A(i,x)\big) \wedge \big(A_B(x,o) \Rightarrow G_B(x,o)\big) \Rightarrow \big(A_C(i,o) \Rightarrow G_C(i,o)\big) \quad (4.4)$$

In other words, the subsystem requirements for $\mathcal{A}$ and $\mathcal{B}$ comply with the overall system requirement for $\mathcal{C}$.

4.6.2 Subservice Decomposition Pattern

The subservice pattern is used in the case when there is only one subsystem that provides the interface of the system and additionally uses the other subsystems to fulfill the requirement. The subservice pattern is illustrated with an example in the following.

Example. To illustrate the subservice decomposition pattern, consider an example from the navigation system, as depicted in Fig. 4.9. One system requirement on the usage service level is:

> "The system proposes a route from the point of departure to the chosen destination."

The interface is described by an input channel *query* and an output channel *route proposal*. The navigation system receives a query for a route proposal issued by the driver and proposes an adequate route from the point of departure to the destination.

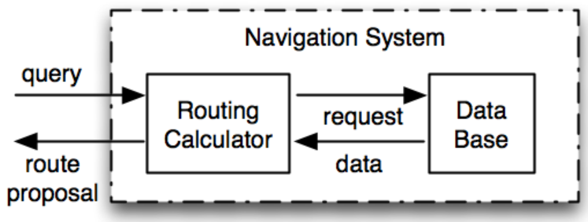


Figure 4.9: Subservice Decomposition of an Example of the Navigation System.

The assumption and guarantee for the navigation system (NS) are:

[8]As defined in [BS01, p. 241], a behavioral refinement relates specifications of the same syntactic interface, where the refined specification may impose further requirements.

- $A_{NS}(\textit{query})$:
  There are valid inputs for the point of departure and the destination.
- $G_{NS}(\textit{query, route proposal})$:
  The system proposes an appropriate route according to the query.

The relevant subsystems on the logical architecture level are the routing calculator and the data base. Their assumptions and guarantees are listed in the following.

- Subsystem *Routing Calculator (RC)*
  - $A_{RC}(\textit{query})$:
    There are valid inputs for the point of departure and the destination.
  - $A_{RC}(\textit{data})$:
    The data base delivers correct information.
  - $G_{RC}(\textit{query,request})$:
    There are valid inputs for the request.
  - $G_{RC}(\textit{data,route proposal})$:
    The system proposes an appropriate route.
  - $A_{RC}(\textit{query}) \wedge A_{RC}(\textit{data}) \Rightarrow G_{RC}(\textit{query,request}) \wedge G_{RC}(\textit{data,route proposal})$
- Subsystem *Data Base (DB)*
  - $A_{DB}(\textit{request})$:
    There are valid inputs for the request.
  - $G_{DB}(\textit{request,data})$:
    The system delivers correct data about the possible routes between the requested points.
  - $A_{DB}(\textit{request}) \Rightarrow G_{DB}(\textit{request,data})$

If the assumption of the data base is fulfilled, the data base can adhere to its guarantee. This again satisfies the assumption of the routing calculator and therefore the guarantee of the routing calculator is met, which means that compliance to the overall system requirement is assured.

In this case, the routing calculator has to make the assumption that the data base delivers correct and up-to-date information and there are valid inputs for the point of departure and the destination. Then it can give the guarantee to propose an appropriate route. The data base is a subservice for the routing calculator which delivers up-to-date information about the possible routes between point of departure and destination.

- Interplay of *Routing Calculator* and *Data Base*:
  - $G_{RC}(\textit{query,request}) \Rightarrow A_{DB}(\textit{request})$
  - $G_{DB}(\textit{request,data}) \Rightarrow A_{RC}(\textit{data})$
- Deduced behavior of the interplay:
  - If the *query* is valid, the *request* is valid.
  - If the *request* is valid, the *data* is correct.

  – If the *data* is correct, the *route proposal* is appropriate.

- Hence, the decomposition complies with the original system requirement "The system proposes a route from the point of departure to the chosen destination." □

The example decomposition for the subservice pattern works. Therefore, the section continues with the universal description of the pattern.

General Description Subservice Pattern. In the case of the subservice pattern (Fig. 4.10), one subsystem interacts at the interface to provide the requested system service while the second subsystem is used as subservice provider by the first one.

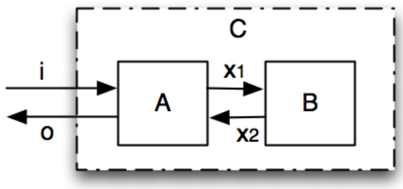


Figure 4.10: Subservice Decomposition Pattern.

The overall system $\mathcal{C}$ in Fig. 4.10 is described by an assumption about input i and a guarantee about output o (same starting point as in Eq. 4.3):

$$A_C(i,o) \Rightarrow G_C(i,o) \tag{4.5}$$

For the subservice pattern, the decomposition results in the following:

- Subsystem $\mathcal{A}$: $A_A(i,x_2,x_1,o) \Rightarrow G_A(i,x_2,x_1,o)$
- Subsystem $\mathcal{B}$: $A_B(x_1,x_2) \Rightarrow G_B(x_1,x_2)$
- Furthermore, the subsystems mutually satisfy each other's assumption about the internal channels x_1 and x_2.

Therefore, the composition of $\mathcal{A}$ and $\mathcal{B}$ is:

$$\big(A_A(i,x_2,x_1,o) \Rightarrow G_A(i,x_2,x_1,o)\big) \wedge \big(A_B(x_1,x_2) \Rightarrow G_B(x_1,x_2)\big) \tag{4.6}$$

Eq. 4.6 is a behavioral refinement of Eq. 4.5, therefore Eq. 4.6 $\Rightarrow$ Eq. 4.5.

By this refinement, the system requirement is appropriately decomposed and refined.

4.6.3 General Decomposition Pattern

The most general case for a subsystem decomposition is that the service a system offers is provided by both subsystems with both of them providing part of the interface.

Example. An example for the general decomposition case is the adaptive cruise control (ACC) system. The *ACCSystem* allows the driver to set a specific speed that the vehicle automatically maintains. The system is depicted in Fig. 4.11.

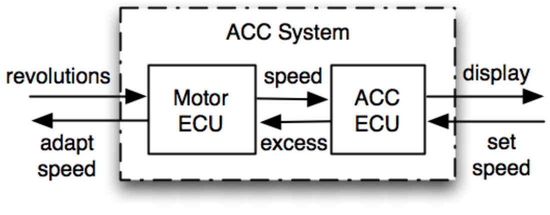


Figure 4.11: Decomposition of an Example from the ACC System.

In this example, the black box interface consists of two input channels, *revolutions* provides the information from the wheel sensors about the current number of revolutions to calculate the current speed, and *setspeed* provides the information from the driver about the speed he wishes to maintain. The output channels are the *display* information that is shown to the driver about the maintained speed and *adaptspeed* is the information that tells the motor to reduce speed in order to obtain the prescribed speed. The assumptions and guarantees of the *ACC System* are the following:

- $A_{Sys}(revolutions, setspeed)$:
  The input *revolutions* delivers the current number of revolutions from the wheels and the input *set speed* delivers the speed request by the driver.

- $G_{Sys}(revolutions, setspeed, display, adaptspeed)$:
  The output *display* delivers feedback for the driver according to the request *set speed* and the output *adapt speed* sends commands to the motor for adapting the speed.

- The system requirement specification for the *ACC System* is:
  $A_{Sys}(revolutions,setspeed) \Rightarrow G_{Sys}(revolutions,setspeed,display,adaptspeed)$

The subsystems are the *Motor ECU* and the *ACC ECU*. The assumptions and guarantees for the system and the subsystems are as follows:

- Subsystem *Motor ECU*

  – $A_{Motor}(revolutions,excess)$:
    The information about current *revolutions* and *excess* speed is available.

  – $G_{Motor}(revolutions,excess,speed,adaptspeed)$:
    The current *speed* is calculated from the *revolutions* and the *excess* information is checked whether it is necessary to *adapt speed* is provided.

- Subsystem *ACC ECU*

- A_{ACC}(*speed,setspeed*):
  The information about the current *speed* is available and the input *set speed* delivers the speed request by the driver.
- G_{ACC}(*speed,setspeed,excess,display*):
  The information about *excess speed* is delivered after comparing the current *speed* to the *set speed* and the feedback is delivered to the driver via *display*.

- Interplay of the two subsystems:
  - *Motor ECU* calculates the current *speed* and sends it to *ACC ECU*.
  - *ACC ECU* calculates for *Motor ECU* whether there is *excess* speed.
- Deduced behavior of the interplay:
  - If the *revolutions* are available, *Motor ECU* delivers the *speed*.
  - If the *set speed* is available, *ACC ECU* calculates *excess* using *speed*.
  - In case of *excess*, *Motor ECU* issues the command *adapt speed* and *ACC ECU* sends feedback to the driver via *display*.
- This complies with the overall *ACC System* guarantee G_{Sys}. □

Hence, the decomposition of the example works as the original system specification of the *ACC System* is fulfilled.

General Description. In the general pattern, both subsystems interact with the surrounding system and provide a part of the interface, as depicted in the white box view of the system $\mathcal{C}$ in Fig. 4.12.

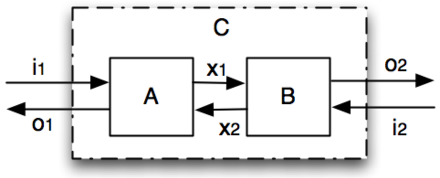


Figure 4.12: General Decomposition Pattern.

For the decomposition, part of the input for system $\mathcal{C}$ goes to subsystem $\mathcal{A}$ via communication channel i_1 and part of the input goes to subsystem $\mathcal{B}$ via communication channel i_2. The output is provided in part by subsystem $\mathcal{A}$ via communication channel o_1 and in part by subsystem $\mathcal{B}$ via communication channel o_2. The subsystems interact via communication channels x_1 and x_2. The specification of the overall system requirement for $\mathcal{C}$ is:

$$A_C(i_1, o_1, i_2, o_2) \Rightarrow G_C(i_1, o_1, i_2, o_2) \tag{4.7}$$

For decomposition, the assumptions and guarantees for the subsystems $\mathcal{A}$ and $\mathcal{B}$ are:

- Subsystem $\mathcal{A}$:
  - Assumptions: $A_A(i_1, x_1, x_2, o_1)$
  - Guarantees: $G_A(i_1, x_1, x_2, o_1)$
  - $A_A(i_1, x_1, x_2, o_1) \Rightarrow G_A(i_1, x_1, x_2, o_1)$
- Subsystem $\mathcal{B}$:
  - Assumptions: $A_B(i_2, x_1, x_2, o_2)$
  - Guarantees: $G_B(i_2, x_1, x_2, o_2)$
  - $A_B(i_2, x_1, x_2, o_2) \Rightarrow G_B(i_2, x_1, x_2, o_2)$
- Furthermore, the subsystems satisfy each other's system-intern assumptions about the channels x_1 and x_2.

Therefore, the composition is a behavioral refinement of the original specification from Eq. 4.7:

$$\begin{aligned}\big(A_A(i_1, x_1, x_2, o_1) \Rightarrow G_A(x_2, x_1, i_1, o_1)\big) \wedge \big(A_B(i_2, x_1, x_2, o_2) \Rightarrow G_B(x_1, x_2, i_2, o_2)\big)\\ \Rightarrow \big(A_C(i_1, o_1, i_2, o_2) \Rightarrow G_C(i_1, o_1, i_2, o_2)\big)\end{aligned} \tag{4.8}$$

In other words, the decomposition according to the general patterns complies with the overall system specification in Eq. 4.7.

4.7 Discussion: Quality Requirements

This section analyzes the decomposition of quality requirements. After giving a definition of quality requirements, the precondition of compositionality is discussed, and then, three handling alternatives for different categories of quality requirements are presented and illustrated with example decompositions.

4.7.1 Definition of Quality Requirements

The following discussion relies on the definitions given by Glinz in [Gli07]:

DEFINITION 4.1 *The set of all requirements of a system is partitioned into functional requirements, performance requirements, specific quality requirements, and constraints.*

A functional requirement *is a requirement that pertains to a functional concern.*

A performance requirement *is a requirement that pertains to a performance concern.*

A specific quality requirement *is a requirement that pertains to a quality concern other than the quality of meeting the functional requirements.*

A constraint *is a requirement that constrains the solution space beyond what is necessary for meeting the given functional, performance, and specific quality requirements.*

An attribute *is a performance requirement or a specific quality requirement. [Gli07, pp.4/5]* □

4.7.2 Precondition for Decomposition: Compositionality

In general, all types of system requirements are treated equally within DeSyRe, no matter whether they are functional or not. The preconditions are that the requirements are refined to a sufficient degree and that compositionality is given, as explained in Sec. 4.4, p. 76.

Specific quality requirements are often difficult to decompose and refine because it requires the expertise of the requirements engineer to find appropriate test criteria or measures for their validation. However, without sufficient validation criteria provided by the requirements engineer, the system designer cannot check whether his specification of the (sub-)system meets the requirements. If a requirement cannot be decomposed, this is an indicator for a necessary refinement before intending a decomposition for the subsystems.

For special quality requirements, the limitation for a guaranteed appropriate decomposition is the compositionality of the subsystem requirements. The hypothesis is that they require additional properties for composition. These additional properties are *not* necessarily known — for some quality attributes, there may be a model for calculation, for others, the property may include probabilities, for some, it may still remain completely unsolved what the additional property is.

However, for a concise representation of the idea and to facilitate the discussion, the hypothetical property is given a name and denoted in a formula: Instead of straight-forward composition of the subsystems as in Eq. 4.2 on p. 78, there is an additional internal property z that specifies the dependency or composition rule[9] for the subsystem requirements specifications S_A and S_B:

$$z \wedge \bigwedge_{a \in S_A} a \wedge \bigwedge_{b \in S_B} b \Rightarrow \bigwedge_{c \in S_C} c \tag{4.9}$$

The property z can represent many different characteristics or relations, also depending on whether the decomposition is performed on the logical architecture level or on the technical architecture level. For example, z can include:

- Probabilities: For example, for calculating composed availability, z can be a model that describes how to include the probabilistic characteristics appropriately.
- Geometric characteristics: In case of distributed systems, z can take account for the geometric characteristics.
- Latency: In case of latency requirements, z can be a calculation model thats adds the necessary latency for composition of components.
- Cable length: For electromagnetic compatibility, cable length is a parameter with high influence. So z can, for example, represent the electromagnetic characteristics of the cable length between two subsystems in a vehicle in order to be able to fulfill a requirement with respect to electromagnetic compatibility.
- Hardware costs: For splitting up a requirement about hardware costs, z could be responsible for taking into account not only the subsystems, but adding the right amount for the communication channels.

[9]The idea of an additional internal property z is inspired by [Bro10].

- Correction algorithms: For example, in systems that use measurements from the environment (e.g. fuel consumption, physics experiments, ...), z can be a correction algorithm.

These are a couple of examples and they do not give an encompassing answer to the question of how to realize z in general. However, for some of the mentioned possibilities for z, there is related work available, with much of it still in progress.

As stated on p. 76, compositionality is still under active research for quality attributes [Can01, PM05, Neuss]. For example, Pavlich-Mariscal [PM05] proposes a composable security definition that uses concern-specific modeling languages. Other quality attributes require predications about probabilities, for example availability. In this direction, Neubeck [Neuss] is working on a model for composing probabilities. For performance requirements, Russell and Zilberstein [RZ91] approach compositionality by using so-called *anytime algorithms* that are characterized by a probabilistic description of the quality of results as a function of time. For composability in service-oriented architectures, see the work by Drugan et al. [DDB+05] on composability of ad hoc mobile middleware.

4.7.3 Decomposition and Alternative Handling of Quality Requirements

Despite the open issues with respect to compositionality for specific quality attributes, the requirements decomposition has successfully been applied to a number of examples. The results do not provide statistical relevance but exemplarily show how specific quality requirements can be decomposed for subsystems. The analysis has been performed according to the categories of quality requirements given by Robertson and Robertson [RR07] that are are also used in the Volere Requirements Specification Template [RR06b].[10] Each category is represented by an example from the case studies of the driver assistance systems [Ris07, FFH+09b].

- Look and Feel Requirements
- Cultural and Political Requirements
- Usability and Humanity Requirements
- Performance Requirements
- Security Requirements
- Legal Requirements
- Operational and Environmental Requirements
- Maintainability and Support Requirements

These categories overlap, but they help in discussing the general handling that applies to most instances of quality requirements. Each of the categories can be assigned a handling alternative that works for all cases that were found when examining examples and discussing with experts on the topic. However, there might be special cases where a specific requirement does not fit into the assignment given below.

[10]The categories do not match exactly with the definition in [Gli07], as performance requirements and special quality requirements are listed together, but this does not restrict the discussion.

Overview of the Alternatives. There are three alternatives for decomposing quality requirements according to their specifics with respect to compositionality (as discussed in Sec. 4.7.2).

1st Alternative: Compositionality of characteristics (as in Eq. 4.2) is given for look and feel requirements, cultural and political requirements, and many usability requirements. These can be decomposed and refined for the subsystems as soon as the responsible system characteristics are identified.

2nd Alternative: Additional rules or models are required to approach compositionality (as in Eq. 4.9) for performance requirements, security requirements, some usability requirements, and some legal requirements.

3rd Alternative: Constraints that are not decomposable in general are issued by operational and environmental requirements as well as maintainability and support requirements, and many legal requirements.

For the first two alternatives, it is important to note that a decomposition often already *includes design decisions* and therefore the subsystem requirements result as *functional* requirements. This is due to having to take a glass box view onto the system in order to decide on a decomposition into subsystems that is presumed as given for the decomposition of the requirements. Such a given decomposition is a design decision and therefore, the decomposition of the requirements may reflect the respective design decision.

Taking a design decision can result in turning the quality requirements into functional requirements for the subsystems. For example when a safety requirement is decomposed by naming what the respective realizing subsystems have to accomplish in order to fulfill the overall system requirement. Many high-level quality requirements can be turned into functional requirements, even before decomposition, by deciding how they should be realized. For that reason, both cases (turning into functional requirements and staying nonfunctional) occur in the examples given below.

1st Alternative: System Properties for Enabling Decomposition

For look and feel requirements, cultural and political requirements as well as many usability requirements, a rule of thumb is proposed as these specific quality requirements are expected to be decomposable and refineable as soon as the responsible system characteristics are identified. The rule for decomposition is:

1. Refine the overall system requirement as precisely as possible according to applicable/responsible properties of the system vision.

2. Decompose the requirement and refine it for the subsystems that exhibit any of these properties.

If all affected subsystems refine the requirement accordingly, compliance with the overall system requirement is guaranteed. For each of the categories, an illustrating example is discussed in the following.

Look and Feel Requirements. These requirements specify the intention of the appearance, and not the detailed design of an interface [RR07, p. 176].

- Example from the RFW system: *The product shall conform to the CI standards.*
  This requirement is still high-level and general. Before performing a decomposition, it is suggested to refine the requirement for the system characteristics that are relevant for the CI.

- Refinement: *The user interface has to provide appearance according to the CI standards.*
  Now the requirement can be decomposed and refined for the subsystems, i.e., every subsystem of the decomposition that realizes part of the user interface has to refine the requirement according to its related characteristics.

- Decomposition (and further refinement): *The display uses typesetting and colors according to the CI standards.*
  In this case, the display is the only subsystem providing part of the user interface and therefore has to refine the requirement, while the RFW controller subsystem is not affected.

Look and feel requirements are expected to be decomposable and refineable as soon as the responsible system characteristics are identified. A rule of thumb for their decomposition is:

1. Refine the overall system requirement as precisely as possible according to applicable (look-and-feel) properties of the system vision.

2. Decompose the requirement and refine it for the subsystems that exhibit any of these properties.

Cultural and Political Requirements. These are special factors that would make the product unacceptable because of human customs, religions, languages, taboos, or prejudices. The main reason for cultural requirements comes when a product shall be sold to a different country [RR07, p. 190].

- Example: *The pictograms of the driver assistance systems have to be intuitively understandable and non-offending all over the world.*
  This requirement can be decomposed and adapted for the subsystems that include pictograms in their user interface, e.g. the RFW.

- Decomposition and refinement: *The pictograms of the RFW system used in addition to the traffic road signs have to be intuitively understandable and non-offending all over the world.*
  Referencing also the road traffic signs prevents the misinterpretation that the user interface substitues all road traffic signs (that differ considerably all over the world) with pictograms.

For cultural and political requirements applies the same notion with respect to decomposition and refinement as for look and feel requirements. The essential step is the identification of the responsible system characteristics that exhibit the relevant properties for realizing the requirement.

Usability and Human Factor Requirements. These requirements make the product conform to the user's abilities and expectations of the usage experience [RR07, p. 178/179].

- Example from DAS: *The driver assistance system shall help the driver to avoid mistakes and thereby reduce the number of accidents.*
  This requirement can be refined for all driver assistance systems. The requirements engineer has to think about what a particular subsystem contributes to the requirement.

- Decomposition and refinement for ACC: *The ACC helps to avoid accidents by automatically reducing the speed of the vehicle in case of too little distance to the preceding vehicle.*
  If every driver assistance system (which are the subsystems in this case) contributes in that manner, the system requirement is met. The system requirement in this example is quite high-level, so one may argue that it has to be concretized anyways to be able to realize it. For that reason, the refinement for the ACC is still high-level as well. The alternative is to first turn the abstract requirement into a concrete one, that can be measured, and then decompose and refine it for the subsystems.

Many usability and human factor requirements are expected to be similar to look and feel requirements with respect to decomposition and refinement. Therefore, a similar rule of thumb applies, in this case for usability characteristics. However, there are some cases, where straight-forward composition does not guarantee compliance with the overall system requirement, for example, when referring to ease of navigation. This example is discussed below within the examples for decomposition with additional properties.

2nd Alternative: Composition Models for Enabling Decomposition

For performance requirements, security requirements, some usability requirements, and some legal requirements, a decomposition that guarantees compliance with the overall system requirement is only possible with adequate composition models. If such a composition model is available, the subsystem requirements can be decomposed and refined, so that their compliance with the overall system requirement is assured as in Eq. 4.9. An example for each category illustrates the idea.

Performance Requirements. These requirements describe specific capacities the product needs to have [RR07, p. 182] like response times, accuracy, reliability, and resource consumption.

- Example: *The response time of the system shall be less than half a second for 90% of system uptime. No answer may take longer than one second.*
  Such requirements can be refined for a subsystem by breaking down the allowed response time for all subsystems and calculating the additional communication overhead between the subsystems. In current practice, these calculations are based on estimations.

- Decomposition and refinement: *The response time shall be less than 0.2 seconds for 95% of system uptime. No answer may take longer than 0.4*

seconds.
Depending on the type of system, the currently used estimations are conservative (worst case) or liberal (average case). Giving guarantees instead of estimations for such runtimes is still under research for real-time system development. In this example, the numbers are not calculated adequately. For precise refinement, a calculus for time and, subsequently, a calculus for probability is needed, which would be captured in the property z introduced in Sec. 4.7.2.

The major problem with performance requirements is that the calculations still rely on estimations. As denoted above, models for compositionality are still under research, e.g. [RZ91, Neuss]. Decomposition and refinement can be performed, but the assurances are insecure.

Security Requirements. Security includes several aspects: confidentiality, integrity, availability, privacy, traceability... Confidentiality means the product's data is not available to anyone except authorized users. Availability means that authorized users are not prevented from accessing the data, and the security devices employed do not hinder or delay the users from getting what they want when they want it. Integrity means that the data held by the product corresponds exactly to what was delivered to the product from the adjacent system [RR07, p. 187/188].

- Example: *The smart remote key system has to prevent unauthorized persons from remotely unlocking the car.*
  This requirement first needs a refinement, that defines how an unauthorized person might try to remotely unlock, before it is possible to decompose it for the subsystems.

- Refinement: *The smart remote key has to prevent that unauthorized persons can systematically guess key codes and transfer them remotely to unlock the car.*
  This definition allows to decompose the original requirement for the subsystems that are concerned with *key codes.* This is a special case that does not yet cover all possibilities of remotely unlocking the car. Further possibilities include trying to spy out the transmitted code. In addition, it has to be noted that in this case the requirement is transformed into a functional requirement by the design decisions that have been taken.

- Decomposition and refinement: *The key control includes a blocker circuit so that in case of 10 consecutive incorrect code words from a remote, it blocks the receipt of all signals (even correct ones) for the next 15 minutes.*
  Thereby, of all the millions of possible code combinations a scanner only has 10 chances every 15 minutes [Cas99]. The original requirement is realized. As overall solution, this is not yet satisfying, but it is a partial solution used in practice as stated in [Cas99]. This subsystem requirement already includes design decisions and turned into a functional one.

Security requirements are often complex in their decomposition and the requirements engineer has to put effort into finding the responsible parameters and system characteristics for realizing the requirement. This usually includes already a certain degree of design decisions with respect to a technical solution which often leads to a transformation into functional requirements. The latter particularity is also noted by Luckey et al. [LFBW10] who map security requirements onto activities in a quality model, thereby usually turning them into functional requirements. If these means are identified, security requirements are expected to be decomposable and refineable. For some security requirements, for example for availability requirements, a composition model is necessary in addition.

Functional Safety Requirements. A particularly important quality attribute for embedded systems that is difficult to assign to any of the categories of Robertson and Robertson [RR07] is safety. Safety requirements play a crucial role in determining the acceptability of a safety-critical system and therefore an example is included in this listing.

- Example: *The electronic stability control (ESC) interferes when the driver is losing control of the vehicle.*
  This real-time requirement first needs a refinement that defines *losing control* before it is possible to decompose it for the subsystems.

- Refinement: *Losing control is defined by either lateral acceleration greater than x m/s, or slip greater than y % (traction control).*[11]
  Thereby, the requirement is turned into a functional one, as *losing control* can now be measured. This definition allows to decompose the original requirement for the subsystems that are concerned either with *lateral acceleration* or with *slip*.

- Decomposition and refinement:
  - *In case of slip greater than y %, traction control has to notify the ESC.*
  - *In case of lateral acceleration (LA) greater than x m/s, the LA sensor controller has to notify the ESC.*
  - *The ESC controller gets activated in case of notification by either traction control or LA sensor controller.*

  Thereby, all subsystems relevant for the activation of the ESC refine the original requirement and their composition leads to compliance.

Safety requirements, like security requirements, are complex in their decomposition and the responsible parameters and system characteristics have to be identified. After that step, safety requirements are expected to be decomposable and refineable.

Usability and Humanity Requirements. The category is listed for a second time to give an example for a requirement where the composition has additional properties that have to be taken into account.

[11]The complete definition of *losing control* involves more parameters and is more complex.

- Example from DAS: *All commands should be fast to reach by interface navigation.*
  This example could either by adopted as is for the subsystems or, as it is still quite high-level and abstract, be refined into a measurable requirement and then decomposed and refined for the subsystems.
- Refinement: *The navigation through the user interface must take no more than 5 steps until the driver reaches the desired command or information.*
  The result is a measurable, functional requirement that can be refined for all driver assistance systems. However, the requirements engineer has to keep track of the different possibilities of navigation within the usage interface under development. The more subsystems share the user interface, the more difficult it will be to overview all paths.
- Decomposition and refinement: *The navigation through the user interface of the RFW must take no more than 3 steps until the driver reaches the desired command or information.*
  Still, this requirement does not guarantee compliance with the system requirement in case of navigation relations with another DAS.
- Handling of the integration: One possible solution is a *central navigation path graph* that depicts the possible cross-DAS navigation paths and therefore allows to check whether the overall navigation still complies with the system requirement.

Many usability and humanity requirements may be decomposed as soon as the appropriate (sub)system characteristics are identified, but some, like the *navigation* example, require the integration of the subsystems to be explicitly included in the composition to guarantee compliance with the system requirement.

Legal Requirements. As the cost of litigation is one of the major risks for software for sale, and can also be expensive for other kinds of software, it is important to be aware of the laws that apply to the developed kind of product. Legal requirements can also have to do with adjacent systems or actors or demand compliance to standards [RR07, p. 186].

- Example: *The driver assistance systems have to comply to the Electromagnetic Compatibility (EMC) [Bun08d] norms.*
  This requirement can be decomposed and refined as soon as the responsible characteristics are identified, for example, for each subsystem that uses a specific kind of short range devices.
- Decomposition and refinement: *The RFW system has to comply to the EMC standard for Short Range Devices operating on frequencies between 9 kHz and 25 GHz.*
  Thereby, the RFW subsystem complies with the overall system requirement, as the relevant characteristic for complying to the EMC norm are the short range devices of the RFW. Further refinement of the requirement has to identify the attributes of the RFW that are affected.

- However, if the subsystems comply to the EMC, this does not guarantee that the logical device connecting them does also fulfill the requirements. In fact, especially long cables (for example, the data buses in cars) have a high impact on the electromagnetic radiation. Therefore, the relation (or connection) between the subsystems has to be considered and documented explicitly in this case to guarantee compliance with the EMC norm.

Many legal requirements can be refined by turning them into technically decomposable safety or security requirements like the *EMC* example, but there are other instances that remain plain constraints. An example for the latter is discussed below within the other constraint examples.

3rd Alternative: Constraint Handling instead of Decomposition

Operational and environmental requirements as well as maintainability and support requirements, and many legal requirements can often not be refined as they are general constraints for the system. For such quality requirements that cannot be refined, an explicit constraint list at the beginning of the subsystem requirements specification (including validation means) is recommended. Examples for the two categories are given in the following.

Operational and Environmental Requirements. These requirements can cover the operating environment, the condition of the user, and partner or collaborating systems [RR07, p. 184/185].

- Example from DAS: *All data has to be exchanged using common message formats.*
  This requirement is high-level and can either by adopted for the subsystems or refined first and then decomposed and refined for the subsystems. The requirement is therefore refined into the following.

- Refinement: *The data of the driver assistance systems has to be exchanged via CAN messages.*
  The CAN bus is one of the data busses in current vehicles and the requirement can be adapted for the subsystems.

- Refinement for the ACC subsystem: *The data of the ACC has to be exchanged via CAN messages.*
  This requirement is adapted for all individual driver assistance systems to ensure their compatibility with the surrounding system.

Operational and environmental requirements can be adapted for subsystems if there is a specific system characteristic that is realized by particular subsystems.

Maintainability and Support Requirements. These requirements capture information about changes that can, to some extent, be foreseen, e.g., in the areas of organization, environment, and business rules [RR07, p. 186]. This type of requirements is also concerned with adaptability, extendability, analyzability, stability and portability.

- Example: *Software updates are provided once a year.*
  This support requirement can not be refined but is rather a constraint.

- Handling: The constraint can be added to a designated list of constraints for the subsystem or be integrated directly into process planning.

The same applies for a number of other examples for maintainability and support requirements that were analyzed. Maintainability is usually realized either constructively, e.g., by proposing guidelines, or assured analytically, e.g., by issuing assessment criteria. A possible solution is an explicit constraint list at the beginning of the subsystem requirements specification.

Legal Requirements. This category is the second one that is listed twice for belonging to two different handling assignments. This time, the requirement is a constraint that can only be adapted for the subsystem, but not decomposed.

- Example: *The use of micro controllers in battery-monitoring sensors has to include a reference to the patent number by Mikrochip. [Mic06]*
  This requirement can only be adapted for the subsystems, but no real decomposition is possible.
- Handling: The constraint can be added to a designated list of constraints for the subsystem.

Legal requirements can either be broken down into safety or security requirements, as in the *EMC* example above, or they persist as plain constraints that have to be adapted and checked manually for validation.

4.8 Tracing

Tracing of requirements is important for a number of purposes, inter alia, project management, validation, and maintenance. Naturally, these tasks are also related to the derivation of subsystem requirements specifications and their management and maintenance.

In the following, *requirements traceability* is understood as defined by Gotel and Finkelstein [GF94]:

> *Requirements traceability* refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through periods of on-going refinement and iteration in any of these phases). [GF94, p. 1]

4.8.1 State of the Art of Tracing

According to von Knethen and Paech [vKP02], there are four core concepts present within tracing approaches: purpose, process, conceptual trace model, and tools. The purposes are characterized by the stakeholders involved in tracing: Customer, project planner, project manager, requirements engineer, designer, verifier, validator, and maintainer.

The process steps are *defining* the traces, *capturing* them, extracting and *representing* the traces, and finally *maintaining* them.[12]

[12]These steps are a constructive approach to requirements tracing; retrieving traces later (post development) is an analytical approach.

Conceptual trace models consist of entities and relationships. Entities are usually characterized by their kind, granularity, and attributes. Relationships occur horizontally on the same level of abstraction as viewpoints and as dependencies, vertically between abstractions as refinements in same and different types of artifacts, and as versioning. Traces have a direction, attributes, an implicit or explicit setting, and can be represented in matrices, graphical models, or as cross references. Von Knethen and Paech distinguish tools in use as general-purpose tools (e.g., spreadsheet), special-purpose tools, workbenches, and case tools [vKP02].

So far, there is no agreement in literature as to which conceptual trace models (in terms of entities and relationships) are necessary to support which stakeholders and tasks. The reason for that lack of agreement is that the conceptual trace model depends on the artifacts to be considered in the domain of application. However, one of the most cited reference models is the one by Ramesh and Jarke [RJ01].

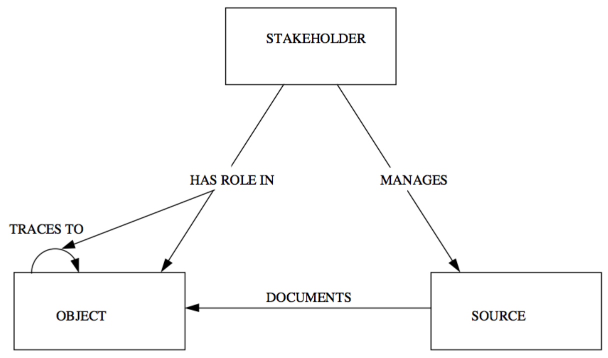


Figure 4.13: Traceability Meta Model by Ramesh and Jarke [RJ01].

Reference Model for Traceability. The meta model by Ramesh and Jarke is based on an empirical study including interviews in 26 software development organizations. The basic meta model consists of the entities *Object*, *Stakeholder*, and *Source* (see Fig. 4.13). The stakeholder manages the sources which, in turn, document the objects. The meta model shall be used to answer the following questions about each entity or relation in the model: *what* information is represented (type of entity or relation), *who* the stakeholders are, *where* in the document sources the information is represented, *how* the information is represented, *why* the conceptual object exists or was modified (rationale), and *when* the information was captured.

Furthermore, Ramesh and Jarke define four traceability link types: *satisfaction*, *evolution*, *rationale*, and *dependency* [RJ01]. Satisfaction and dependency are product-related while rationale and evolution are process-related traceability links. A dependency link is explained by a satisfaction link and an evolution link is explained by a rationale link.

The model is the basis of the partial model described for this work.

4.8.2 State of the Practice of Tracing

The following three reports constitute a representative overview of the state of practice in requirements tracing. Gotel and Finkelstein [GF94] report on a questionnaire study with more than 100 practitioners, Ramesh [Ram98] reports on a study with 138 practitioners, and Eyged et al. present their lessons learned from three case studies on a value-based approach [EGHB09].

Gotel and Finkelstein [GF94] describe the current support for requirements traceability in terms of basic techniques and automated support, analyze why there is still a requirements traceability problem (lack of a common definition and conflicting underlying problems), propose a framework, and suggest a research agenda that emphasizes pre-requirements specification traceability.

Ramesh [Ram98] analyzes findings from a comprehensive survey in order to identify what motivates users to employ traceability practices. Critical factors that influence the practice of requirements traceability are general conditions like recognizing and articulating a traceability problem, the usage of traceability by developing methods and acquiring or developing tools, and changes in system development policies.

Eyged et al. [EGHB09] conclude from their case studies that trace acquisition "requires broad coverage but can tolerate imprecision". Therefore, they suggest a traceability strategy that provides trace links quickly, refines them according to stakeholder-defined value considerations, and supports later refinement (for changes within the value considerations).

The conclusions of these reports on the state of practice are considered for the proposed tracing approach for DeSyRe in terms of providing suggestions for low cost, easily adaptable options for requirements tracing.

4.8.3 Proposed Tracing Approach

The tracing between the requirements can either be established constructively during specification or analytically on demand. The constructive approach leads to complete capturing of all traces during the decomposition and refinement of the requirements. At the same time, the maintenance effort for a complete tracing can be cost-intensive if performed manually.

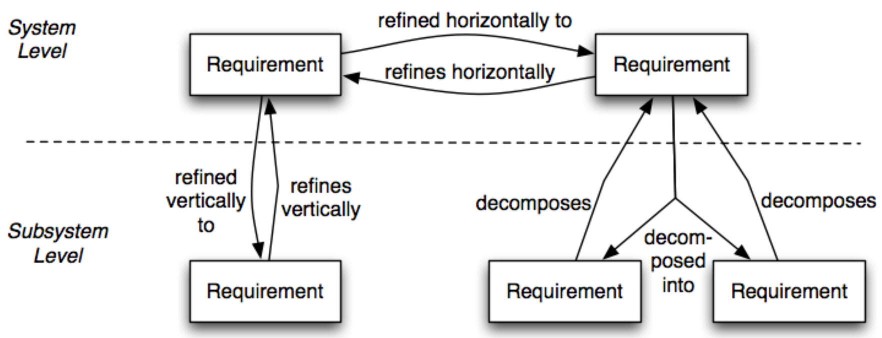


Figure 4.14: Tracing Model for DeSyRe.

Constructive Approach. The requirements' traces, including the respective type of relationship, are documented while decomposing and refining the requirements. The basic model for the tracing documentation is depicted in Fig. 4.14. A *Requirement* on the system level can be either:

- refined horizontally by another requirement on the system level (in case it does not yet provide sufficient information to be transitioned into a requirement on the subsystem level),
- refined vertically by a requirement on the subsystem level (in case there is only one subsystem concerned with the requirement),
- or decomposed into a number of subsystem requirements.

In relation to Ramesh and Jarke's meta model (Fig. 4.13), the entity *Requirement* can be interpreted as a type of *Source* and the relationships are all of the type *evolution*.

The traces can be documented using a variety of techniques, for example, reference indexes, traceability matrices, or templates [GF94]. To find the optimal strategy, different tracing strategies may be compared by use of the tracing activity model TAM as proposed by Heindl and Biffl [HB08, HB07], but further discussion of tracing strategies is out of scope for this work.

The dependencies have to be maintained across refinement and decomposition. For example, considering a system requirement X_C that depends on another system requirement Y_C:

- X_C is refined to subsystem requirement X_A.
- Y_C is decomposed into subsystem requirements Y_A and Y_B.
- Consequently, X_A depends on Y_A and Y_B.

Documentation and maintenance of the requirements traces and their dependencies is crucial for successful change management.

Analytical Approach. An analytical approach can either be performed on demand or if, after inadequate change management, traces have to be recovered. Approaches to trace recovery are, for example, information retrieval algorithms [LO10], often supported by tools like RETRO [HDS$^+$07], or latent semantic indexing [LD06].

For a better retrieval rate, different trace recovery tactics can be combined as each model might miss certain relations, e.g., a keyword-based approach might miss a relation where less frequent synonyms have been used.

Implications of Trace Orders. Different decomposition alternatives of a system lead to different requirements decompositions and, therefore, to different requirements trace graphs or trace orders. For a given decomposition, there may be alternatives for restructuring the requirements decomposition to a certain extent while maintaining the dependencies. The aim of restructuring the decomposition could be to retrieve an easier maintainable trace order, for example, if maintainability is a key concern for the system under development.

Consequently, the question of what are good ordering criteria (in terms of classification) for requirements to support architectural design arises. This issue is partly addressed in Sec. 3.6 but worth further, future investigation in order to support requirements analysis.

Summary. This chapter first introduced a model of a subsystem based on the system model of Broy et al. [BSW+08] with three abstraction levels and discussed the distribution and description of a subsystem across these abstraction levels.

Then, the chapter presented a systematic approach to the decomposition of system requirements into subsystem requirements for a given system decomposition. Case differentiation was made according to whether a requirement could be refined so that it would include a concrete reference to a specific part of the system. This concreteness, plus compositionality, are the prerequisites for expressing a requirement in assumption/guarantee style that can be decomposed.

It was detailed how requirements decomposition and refinement can be performed using three decomposition patterns, two special cases that already apply for most requirements, and the general case for the rest of them. Each of them was illustrated with an example and then described in general. It was shown how, after decomposition into subsystem requirements, the compliance to the original system requirement is fulfilled.

Furthermore, the decomposition and refinement of quality requirements were discussed. These can either be decomposed as soon as the responsible system characteristics are identified, or an additional composition model is needed, or they issue constraints that have to be acknowledged. Thereby, the presented categorization is not strict, there are example requirements that fit into different categories and categories that exhibit example instances for different handling alternatives.

Finally, a simple model for tracing between system requirements and subsystem requirements is proposed. The application of requirements decomposition and refinement in combination with the system decomposition (presented in Chap. 3) within a surrounding software development process is shown in the subsequent chapter.

Chapter 5

The DeSyRe Method - Decomposition of Systems and their Requirements

Contents

Axel van Lamsweerde stated: "...specification technology needs to provide CONSTRUCTIVE methods for specification development, analysis, and evolution." [vL00] Therefore, the method De*composition of* Sy*stems and their* Re*quirements* (*DeSyRe*) is presented as practical guideline. It integrates the decomposition of a system from Chap. 3 and the refinement of its requirements in the transition from system to subsystem (Chap. 4) in order to systematically derive a subsystem specification. For the method at hand, the emphasis lies on describing it constructively for high applicability.

This chapter first discusses related work, next presents an overview of the *DeSyRe* method, then explains the starting point of the method and, subsequently, each of the method's phases. The single steps are illustrated with a running example from the driver assistance systems (DAS) introduced in Sec. 2.6.

5.1 Related Work for Artifact-based Transition from Requirements to Architecture

There is a number of approaches to perform a systematic transition from requirements to architecture on the basis of artifacts. However, all these works specialize on certain notation techniques, while this approach does not imply a specific notation technique to be used.

KAOS Goal modeling. Brandozzi and Perry [BP01] describe briefly how to formally derive an architecture in Architecture Prescription Language (APL) notation from goals in KAOS notation ("Knowledge Acquisition in autOmated Specification"). It inspired van Lamsweerde, the original author of the KAOS notation, to describe the way from system goals to software architecture using an event-based architectural style [vL03]. This works as long as plain pattern matching is applicable, but the author already states in his conclusion that the given constructive guidance is "insufficient in a number of situations where architectural features need to be propagated bottom-up" [vL03, p. 38].

In contrast, DeSyRe includes a wider range of view points and information sources with the decomposition criteria catalogue and the artifact model. Information that often needs to be promoted bottom-up is included especially in the technical criteria of the decomposition catalogue and in the artifacts of the context of the hardware/software abstraction level of the artifact model.

Patterns from NFRs. Eric Yu proposes a goal-oriented requirements (GRL) language, which is intended to capture especially non-functional requirements (NFRs) in patterns [GY01a]. GRL does this with a scenario-oriented architectural notation called Use Case Maps (UCM) to model requirements and architecture incrementally [LY01, GY01b].

The approach limits itself to the NFR part that can be captured within goal modeling (thereby also omitting bottom-up promoted requirements). In contrast, DeSyRe aims at the full range of available requirements and information sources for guiding the way through requirements engineering to design.

CBSP intermediate language. The "Component, Bus, System, and Properties" (CBSP) language, proposed by Boehm, Egyed, Grünbacher, and Medvidovic [MGEB03, EGM01, MBE01], is an intermediate language for representing requirements in an architectural fashion, thereby representing an incomplete architecture. The approach starts with a (potentially incomplete) set of requirements, identifies potential architecture elements by applying a taxonomy of architectural dimensions, and results in an intermediate model that captures architectural decisions. It is integrated with the EasyWinWin negotiation process and is partially tool-supported.

The CBSP bridges a large gap by linking directly from requirements to elements of the technical architecture, like busses. Instead, DeSyRe includes an abstraction level for logical architecture that is independent of hardware details. Furthermore, CBSP uses the coarse-grained "negotiation rationale view" from the WinWin approach to capture the requirements, where "stakeholder objectives and goals are expressed as goals" [MGEB03, p. 201]. Instead, DeSyRe captures requirements from a broader range of information sources including bottom-up technical constraints.

ATRIUM architecture model. Developed in the research project MetaSIG, Navarro et al. [NRP03, Nav07] offer the process ATRIUM, based on the architecture model PRISMA. ATRIUM stands for "Architecture generaTed from RequIrements applying a Unified Methodology". The intention is to guide the architecture development from the inception and handle changes of requirements over time using aspect-orientation. This is done by defining goals and operationalizing architectural scenarios.

The main concerns of ATRIUM are maintainability and adaptability of software artifacts by use of aspect-orientation [Nav07, p. 71]. In contrast, the aim of DeSyRe provides concrete methodical guidance based on an artifact model.

ITEA EMPRESS project. Paech et al. [PvKD$^+$03] propose an experience-based approach for integrating architecture and requirements engineering in the context of the ITEA EMPRESS project. The method shows how different kinds of experience-based artifacts, such as questionnaires, checklists, architectural patterns, and rationale, can beneficially be applied.

EMPRESS does not consider explicit abstraction levels or a detailed artifact model. The findings of the experience-based approach served as additional information to the reports of the practitioners we questioned in our interviews for the development of the artifact model.

COSMOD-RE approach. Pohl and Sikora [PS07a] describe COSMOD-RE that supports the co-design of requirements and architectural artifacts. The authors describe architectural scenarios and aim for a first coarse-grained sketch of the system architecture composed by logical components.

However, COSMOD-RE does not give concrete guidance on how to proceed to decompose a system and derive these architectural artifacts. In contrast, DeSyRe explicitly emphasizes the importance of practical guidance.

All of these approaches specialize on a certain type of artifacts. In contrast, the transition described in DeSyRe abstracts from specific artifacts and instead describes a general transition from system to subsystem requirements.

5.2 Outline of the DeSyRe Method Phases

The approach is an application guideline that is derived from the preceding chapters about the decomposition criteria analysis and the requirements refinement.

Artifact-orientation. An artifact-centered approach is considered more valuable for strongly distributed software development as present in the automotive domain than prescribing smallest process steps.

A system with widely distributed development includes many development teams and therefore is hardly likely to be developed using the exactly same processes and tools. This leads to the pragmatic solution of defining a concrete artifact model with assigned contents, but a less strict process for how to develop the artifacts.

Apart from an easy integration into existing processes, further advantages of defined artifacts are that they can serve as basis for discussions and contracts, and their quality and progress can be checked.

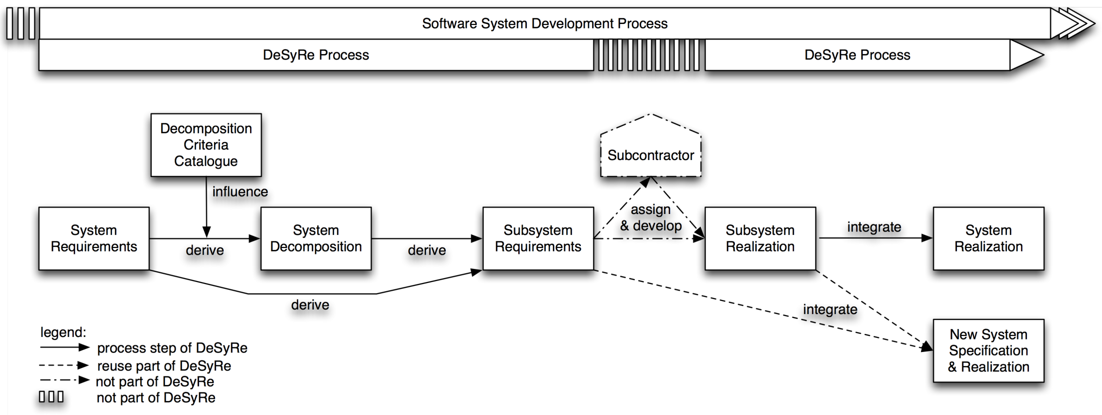


Figure 5.1: Process DeSyRe

Method Phases. The method DeSyRe describes the decomposition into subsystems and the deduction of separate subsystem requirements specifications. The purposes of the decomposition are, as depicted on the right hand side of Fig. 5.1, distributed development and subsequent integration as well as reuse of subsystems.

The process includes the following phases:

- *Decomposition into subsystems* (Sec. 5.4): DeSyRe begins after an initial set of System Requirements has been elaborated (left-most box in Fig. 5.1). By help of the Decomposition Criteria Catalogue (up-most box in Fig. 5.1), the System Decomposition is derived (subsequent box).

- *Transition to subsystem requirements* (Sec. 5.5): The System Requirements and the System Decomposition are the input for deriving the Subsystem Requirements (at the center of Fig. 5.1).

- *Delivery of subsystem specification* (Sec. 5.6): At that point of the development process, the subsystem specification (*Subsystem Requirements* in Fig. 5.1) is delivered to the subcontractor (house shape in Fig. 5.1) who subsequently develops and then delivers the assigned subsystem (*Subsystem Realization* in Fig. 5.1).

- *Integration* (Sec. 5.7): Finally, the subsystem is integrated in the *System Realization* or reused in a *New System Specification & Realization.*

In the following, the starting point of the method and, then, each of the phases will be described in detail.

5.3 Starting Point: Required Artifacts

The point of origin (see Fig. 5.2) for the decomposition of a system into subsystems and, thereafter, the specific concentration on individual subsystems, is a partial instance of the artifact model.

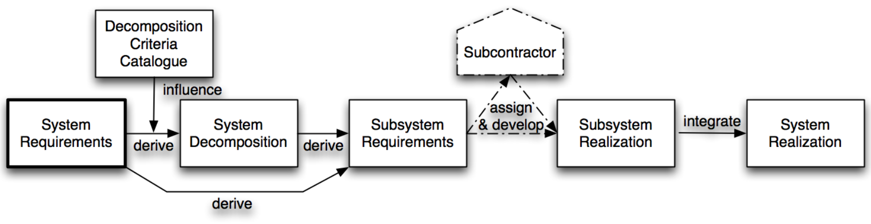


Figure 5.2: Starting Point of Process DeSyRe

The requirements engineer has to elaborate the black box specification of the usage service level of the artifact model, i. e., the respective context, requirements, and design artifacts, as depicted in Fig. 5.3. These artifacts are the required ones to start with DeSyRe, because they represent the usage service level of the overall system (as presented in Sec. 2.5.3) and are elaborated before conducting the initial system decomposition.

The responsible author for these artifacts is the system development organization as producer of the overall system. More precisely, the authorship lies with their requirements engineers who, in turn, have to communicate with the other stakeholders to assess all relevant information, as depicted earlier in Fig. 3.2.

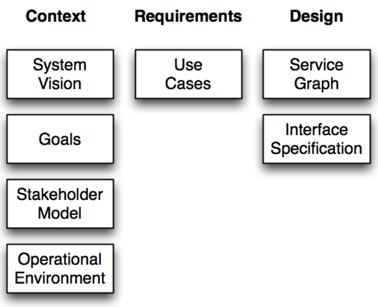


Figure 5.3: Required Artifacts for Decomposition.

Running Example. For illustration, here are two of the artifacts for the driver assistance systems, namely the system vision and the use cases. The other artifacts from the category context, namely Goals, Stakeholders, and Operational Environment, are needed to ensure completeness of the requirements elicitation activities (see App. 6.1.4).

System Vision: “A well-interrelated network of driver assistance systems shall support the driver in demanding traffic situations and thereby increase the safety of the vehicle."

The **Use Cases** were identified in brainstorming sessions and derived from the goals with BMW employees after identifying the system vision and the goals. For a more encompassing list, see App. 6.1.4.

- The driver is about to change lanes. The system displays a vehicle in the blind spot.
- The driver is using cruise control and approaches a vehicle ahead. The system decelerates.
- The driver is driving 100km/h although there is a speed limit of 80km/h. The system displays a warning.

In the design category of the artifact model, a Service Graph helps making dependencies explicit between usage services that realize the use cases. From this basis of artifacts, the software architect can proceed with the decomposition as described in the following sections.

Quality Assurance: Dependency Analysis. An optional but recommended step is an analysis of requirements dependencies, influences, constraints, and potential conflicts. Thereby, the requirements engineer assures the quality of the requirements. What makes such an analysis challenging is that the relations between requirements are often not documented explicitly. Examples for this are shared input or output channels like hardware sensors, displays, etc. [Res08]. Further discussion on implicit knowledge can be found in the thesis of Feilkas [Fei10].

One possibility to perform such an analysis is by help of the work by Herrmann, Paech, and Plaza [HPP06]. The authors define a list of requirements dependencies like refinement, feature bundle, or architectural restriction [HPP06, p.29]. Oftentimes, more than one relationship applies at the same time and that there are also relations between the relationships of the model. However, it is not crucial to mark all possible dependencies for the realization of the system, but it is important to find them as they give hints on potential requirements inconsistencies or conflicts.

Within the same work, Herrmann, Paech, and Plaza [HPP06, pp. 7,8] also define three types of requirements conflicts, namely inconsistency, contradiction, and feasibility conflict. No matter which type, the conflicts have to be resolved before the actual decomposition.

The step *dependency and conflict analysis* is optional because it does not necessarily change the requirements in a way imposed by DeSyRe. However, it is strongly recommended to perform such a step of quality assurance as consistent requirements with explicit dependencies are the basis for successful systems' development. Therefore, we assume for the following that conflicts are resolved and dependencies denoted explicitly.

5.4 Decomposition into Subsystems

For this step, the requirements engineer has to deliver the elaborated artifacts of the usage level to the system architect who is responsible for the decomposition.

The division into subsystems is based on the reference catalogue of criteria that has been defined and explained in Chap. 3. The steps for the decomposition are, first, a consideration of the reference catalogue, then, a relationship analysis of the system requirements, and finally, the division into subsystems.

5.4.1 Consideration of the Reference Catalogue

The reference catalogue of decomposition criteria from Chap. 3 serves as reference guide suitable for different project situations. Therefore, it is necessary to analyze the relevance of the given criteria for the project at hand (Fig. 5.4).

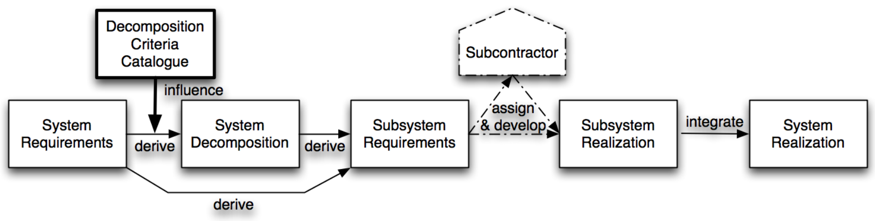


Figure 5.4: Consideration of Reference Catalogue in Process DeSyRe

A customization of the catalogue is optional. The catalogue serves as growing knowledge base and can be amended either during its first use in a company, or if experience in a conducted project brought forward new criteria.

If the requirements engineer or the system architect decide to make alterations to the catalogue, it is necessary to analyze the new relationships and possible influences between the criteria. The relationships between already identified criteria were discussed in Sec. 3.7.

Finally, priorities have to be chosen considering the optimization factors (as described in Sec. 3.2.1) which are most important for the system to be developed. These are usually given by the high priority business goals that are captured in the artifact *Goals* defined in Sec. 2.5.3.

The instance of the catalogue makes implicit knowledge explicit and helps to take decisions on the basis of a seizable artefact with visible-made experience. The information needed for the decomposition is gathered via the description template described in Sec. 3.2.3, which was filled in for each criterion in the catalogue in Sec. 3.3 – Sec. 3.6.

Running Example. The decomposition criteria from the reference catalogue are instantiated and described for the driver assistance systems. Two examples, the criteria Economics (from the category *Directive Criteria*) and Reliability (from the category *Quality Criteria*), are given here; the complete list of the instantiation can be found in Sec. 6.1.4.

Economics

- Reuse: All of the driver assistance systems have already been developed in the previous vehicle series, so reuse is strongly expected.

- Cost Models: The cost of the hardware parts is crucial as even cents sum up critically for the number of produced vehicles. Software development costs are considered less critical, but in this case is connected with reuse, as a decomposition for high reusability will decrease the costs for the next iteration.

Reliability. Maturity, recoverability, and fault tolerance are crucial for all vehicles participating in road traffic. For system decomposition, this can mean that certain subsystems are grouped onto redundant electronic control units.

5.4.2 Decomposition Realization

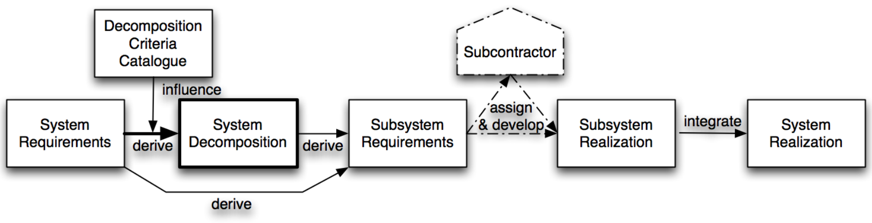


Figure 5.5: Decomposition Realization in Process DeSyRe

After eliciting the most relevant criteria, the system architect decomposes the system accordingly, see Fig. 5.5. In general, it is assumed that there are two main ways that do not necessarily exclude each other, but also work in concert, see Fig. 5.6: Either there are architectural criteria that have to be obeyed and that predetermine the decomposition, or if not, it is to be chosen whether it is more appropriate to decompose the system according to functional or quality criteria. The directive criteria will mainly have an indirect influence that give more general guidance.

The procedure for decomposing the system according to the introduced criteria is performed in the following major steps:

1. Identify architecturally significant requirements.
2. Review catalogue, analyze and assign the requirements to the given categories.
3. Consider and apply the criteria according to the order depicted in Fig. 5.6 in a logical blueprint either on the basis of features or on the basis of architectural constraints.
4. Assure quality of the blueprint.

These steps are explained in more detail to allow for concrete guidance during systems development:

1. Identification. The preparation for the decomposition is to identify the requirements that qualify as architecturally significant in each category. Jazayeri, Ran, and van der Linden give suggestions for detecting them [JRvdL00, p. 11]:

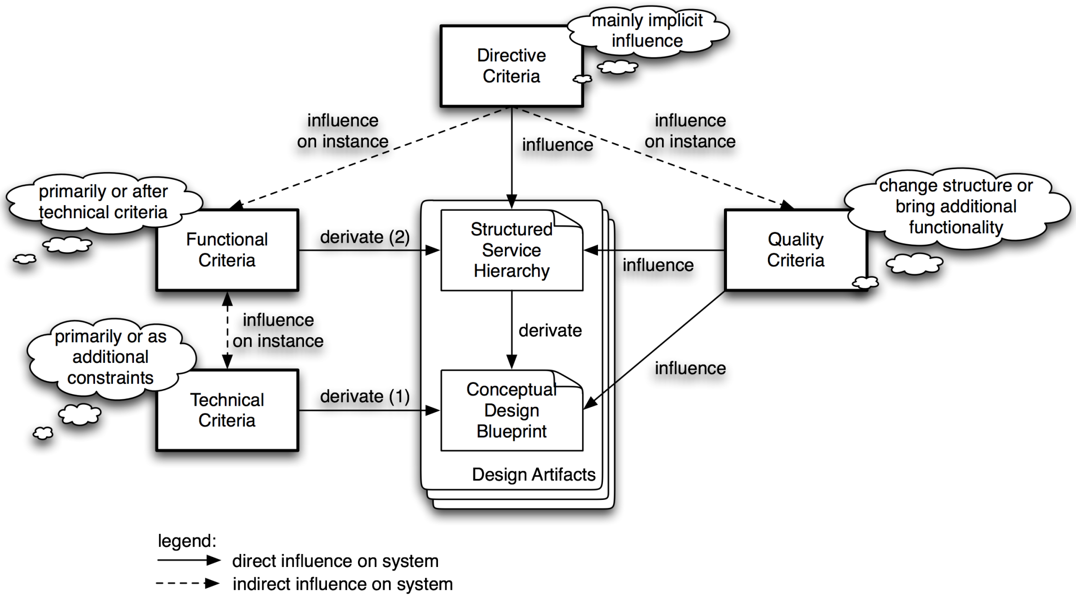

Directive Criteria
mainly implicit influence
influence on instance
influence
influence on instance
primarily or after technical criteria
Functional Criteria
derivate (2)
Structured Service Hierarchy
influence
Quality Criteria
change structure or bring additional functionality
influence on instance
derivate
influence
primarily or as additional constraints
Technical Criteria
derivate (1)
Conceptual Design Blueprint
Design Artifacts
legend:
direct influence on system
indirect influence on system

Figure 5.6: Criteria and Application

- "Requirements that cannot be satisfied by one (or a small set of) system components without dependence on the rest of the system. (...)"
- "Requirements that address properties of different categories of components (...)."
- "Requirements that address processes of manipulating multiple components (...). "

2. Analysis. After their identification, these requirements have to be analyzed and assigned to the adequate criteria categories *directive* (Sec. 3.3), *functional* (Sec. 3.4), *quality* (Sec. 3.5), and *technical* (Sec. 3.6). Within each category, they have to be ordered according to their impact or priority. A requirement may belong to more than one category, however, it is not helpful to introduce redundancy. Instead, the requirement should be assigned to the one category that the system architect perceives to be the most important for the specific content.

3. Application. The (partial) order of the criteria's consideration and application during the design process is depicted in Fig. 5.6. The starting point are the directive criteria, which influence the quality and functional criteria. The latter again influence the technical criteria.[1] There are four theoretic alternatives, inter alia depending on whether there are architectural constraints that in fact predetermine the architecture or not.

[1]These are only the general influences. Additionally, there may also be examples where directive criteria influence technical criteria and where quality criteria influence technical criteria.

Review the catalogue and check for more information related to the criteria within the requirements artifacts.

(a) Technical Criteria Predominance. If there are architectural constraints, the architecture is already predefined and the functional and quality criteria can only be considered secondarily. This directly imposes or at least can restrict the logical design, a coarse-grained blueprint of the system's architecture, leading to derivation (1) in Fig. 5.6. If the system architect finds that there is a predominance of technical criteria that leads to designing the system similarly to previous systems, he should perform an architecture evaluation to determine the appropriateness of the old architecture and thereby justify his decision and assure design quality.

(b) Functional Criteria Predominance. In the alternative case, when there is no such predetermination, the architecture is organized according to the functionality of the system. This allows us to derive a structured service hierarchy with user-perceptible functions decomposed into realizable subfunctions, depicted by derivation (2) in Fig. 5.6. An analysis identifies parts of functionality that are alike within the different use cases and functional requirements of the system. Those common or alike parts are then abstracted to, grouped, and realized as logical components.

In general, (a) leads to a less abstract decomposition than (b), because it has a stronger connection to technical constraints, but both solutions primarily lead to a logical blueprint of the architecture.

Within the directive criteria, especially in big companies, the organizational criteria (e.g., in terms of Conway's Law, cf. Sec. 3.3.1) take a significant influence on the decomposition of complex systems. The reason for this is that the departments are often already organized according to special application domains, for example, in the automotive domain, there is usually a carriage department, engine control department, driver assistance systems department, etc. However, according to whether the organization is structured according to functional system units or technical system units, the predominance will be functional or technical, and the respective above case applies.

4. Quality Assurance After an initial decomposition, either according to architectural or functional criteria, the quality criteria concerning architecture are evaluated (see also evaluation methods according to Bengtsson and Bosch [BB98]). In case they are not sufficiently realized by the initial decomposition, the system architect has to reconsider and modify the design accordingly.

The quality criteria can influence the architecture of the resulting system in two ways: Either the structure is modified according to a certain principle, e.g., maintainability, or some functionality is added to fulfill the requirement, e.g., a component for user identification to satisfy a security requirement. For realizing specific quality requirements, please refer to, e.g., the NFR framework from Chung et al. [CNYM00] and the architectural techniques described by Bass et al. [BCK03].

Defining the Architecture. Furthermore, the following rules of thumb for defining an architecture, according to Bass et al. [BCK03, p. 16], are widely accepted in academia and industry:

- Well-defined modules use the principles of information hiding and separation of concerns for their functional responsibilities.
- Well-defined interfaces per module encapsulate changeable aspects, so the development teams can work independently.
- Well-known *architectural tactics* [BCK03, Chap. 5] help to achieve quality attributes.
- Never depend on a particular version of a tool or other commercial product.
- Modules that produce data should be separate from modules that consume data.
- Parallel-processing systems need well-defined processes or tasks.
- Every task or process should be written so that its assignment to a specific processor can easily be changed.
- Use a small number of simple interaction patterns (= do same things in the same way throughout).

These rules of thumb are often known in a different wording, for example as separation of concerns with respect to separating responsibilities, separating technical aspects and application domain, and using black boxes. Further principles to keep in mind are reusability, maintainability, and other quality attributes, as well as the so-called KISS principle (Keep It Simple and Stupid) that captures the idea to choose the most simple feasible approach for a solution when there are a couple of otherwise equal possibilities.

Running Example. For the driver assistance systems, the conclusion is to apply a decomposition primarily according to the identified services, the user-perceived functionality. One usage service cluster is realized by one logical function group or logical component as depicted in Fig. 5.7.

Consequently, the driver assistance systems are decomposed into the subsystems Blind Spot Detection, Lane Departure Warning, Adaptive Cruise Control, Radio Frequency Warning, Night Vision, and Driver Drowsiness Detection.

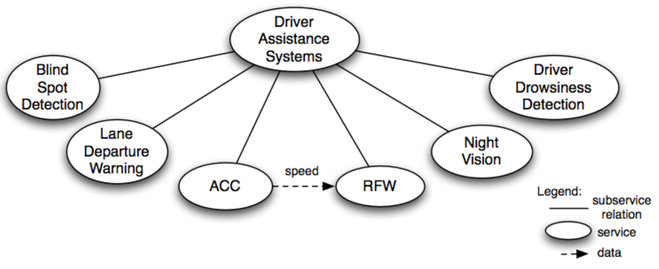


Figure 5.7: Service Graph of DAS.

5.5 Transition to Subsystem Requirements

The general idea of transition from system requirements to subsystem requirements is discussed and analyzed in Chap. 4. The applicable patterns are presented and illustrated with examples in Sec. 4.6.3. For some requirements engineering artifacts, the application differs somewhat from the plain adoption of the patterns. Therefore, the artifacts are discussed separately in the following.

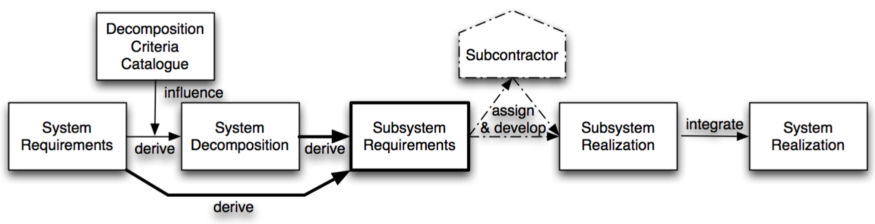


Figure 5.8: Transition to Subsystem Requirements in Process DeSyRe

In Sec. 2.5.3, it was stated that the artifacts of the content categories Context and Requirements may be refined on the lower levels if necessary, but their general form would stay the same. For the transition of the point of view of the overall system to the point of view of a subsystem, such a refinement is necessary. This section describes the transition to subsystem requirements (Fig. 5.8).

Thereby, the transition from system level to subsystem level differs for specific types of artifacts as detailed in the following.

5.5.1 Context

A systematic derivation and refinement of requirements for subsystems deduces a system vision, refines the goals, deduces the stakeholders, and refines the operational environment. However, as the subcontractor is usually not intended to know all the information contained in those artifacts, some of the information (e.g., business goals) may intentionally be omitted in the subsystem requirements specification.

System Vision. The system vision is deduced from the goals of the overall system and the initial decomposition, as the system vision in general is not detailed enough to directly derive system visions for the subsystems.

Running example. The system vision of the driver assistance systems ("A well-interrelated network of driver assistance systems shall support the driver in demanding traffic situations and thereby increase the safety of the vehicle.") provides a rough idea, but does not allow to deduce the subsystems. Instead, knowledge of goals and and decomposition allows to deduce subsystem visions, e.g., for the Radio Frequency Warning System: "RFW supports the driver in coping with the information input from the surrounding environment by use of radio frequency signals."

Goals. The goals on the subsystem level are decomposed and refined for the subsystems according to the given initial decomposition. For each subsystem, the list of goals is walked through and analyzed and documented how the subsystem can contribute to the system goal.

Running example. The driver assistance systems' goal "Provide active safety" can be concretized for the RFW subsystem to "Provide active safety by preventing that the driver misses a traffic sign which could cause an accident".

Stakeholders. The stakeholders on the system level and on the subsystem level are not necessarily the same, but there is strong overlapping. Some of the stakeholders might be especially involved with a particular subsystem, but there is no new information about them on the logical level, the stakeholders do not change according to the degree of detail. Other stakeholders might not be involved at all with specific subsystems, instead, there may be others that were not visible yet on the system level.

Running example. The driver assistance subsystems as black box do already include suppliers in their stakeholder list. However, on the subsystem level, there may be additional suppliers for the realization of a specific subsystem, e.g., for special hardware, because the design choice had not been taken yet.

Operational Environment. In contrast to the business-related artifacts, the appearance of the Operational Environment does change according to the degree of detail on the logical level.

The point of view of the requirements engineer shifts from interacting surrounding systems to interacting surrounding subsystems and possibly also to surrounding systems, i.e., the developer zooms in on a particular subsystem. This shift means that, on one hand, there is a refinement of the description of those surrounding systems that interact with the subsystem under analysis and, on the other hand, other subsystems of the overall system under development turn into being subjects of the context of the subsystem under analysis on the logical level. The system environment on that layer can be divided into a technical and a physical aspect, as defined by Weyer [WP08, pp. 171/172]. For each of them, the documented elements are subject-matter, event, input parameter, and output parameter.

Running example. Before, the driver assistance systems were the black box and the surrounding vehicle and the road traffic were the operational environment. For the RFW system, the operational environment consists of the parts of the vehicle that it interacts with, e.g., the display (to inform the driver), and the parts of the road traffic that it receives input from, e.g., traffic signs.

5.5.2 Requirements

For the requirements artifacts, the situation is different from the context artifacts. In this content category, all artifact types may be refined or derived from the respective usage level artifacts.

Use cases. For the vertical refinement[2] of use cases, it is possible to distinguish between two types of scenarios as defined by Sikora [BGL$^+$08, pp. 17/18]. One type is *local scenarios*, which represent only system-internal interaction, and the other one is *system-wide scenarios*, which document interaction at the external system interface. Mixed forms of the two types also occur.

Scenarios with interaction at the system interface are the ones which are already present on the usage level. These are now refined on the logical level and the actors within the respective scenarios are the (equally refined) *subject matters* of the physical and technical context in the Operational Environment. Therefore, these subject-matters belong to the surrounding systems on the usage level.

System-internal scenarios are not yet documented on the usage level as they are still encapsulated through the black box representing the system under development. These are captured for the first time on the logical level and the actors are those subject-matters of the physical and technical context that were not yet present on the usage level either, in explicit, of the adjacent subsystems. The general pattern from Sec. 4.6.3 can be used for guidance on the decomposition.

Running example. For the RFW system, a system scenario is any concrete scenario instantiated for one of the use cases, for example:

> "The RFW system reads a speed limit traffic sign and displays a warning for the driver, because he is driving too fast."

For the system level scenarios, the RFW system is still black box. On the subsystem level, the scenario is decomposed into two scenarios for the subsystems (as depicted in the example in Fig. 4.7):

> "The RFW control reads and processes a speed limit traffic sign and sends the information to the display."
> "The RFW display receives information about speeding and shows a warning for the driver."

Requirements. An equivalent systematic refinement of the requirements leads to more detailed information about function, data, and behavior on the subsystem level. The functions thereby are performed only by the subsystem under analysis and the occurring actors are equal to the ones present in the scenarios on the logical level. Consequently, there are requirements from the usage level, which have to be refined, and new requirements that are derived from the new scenarios on the logical level.

For the requirements that are to be refined, the description of requirements refinement with assumption / guarantee specifications is applied. The general procedure is a case differentiation and then application of the respective pattern (see also Sec. 4.4, depicted in Fig. 4.5). There are three patterns: two special cases that already apply for many requirements, namely pipeline (Sec. 4.6.1)

[2]Refinement can occur in two types: Horizontal refinement stays on the same abstraction level, vertical refinement makes the transition to the next abstraction level. In the case at hand, there is a transition to the next lower abstraction level, the logical level.

and subservice (Sec. 4.6.2), and a general pattern that can always be applied (Sec. 4.6.3).

Sec. 4.5.2 that exemplarily describes the refinement in case of an *1:m* transition for a requirement that has to be refined for a number of m subsystems.

Running example. An example for each pattern was already given in Sec. 4.6: Adaptive Cruise Control for the general pattern (see Fig. 4.11), Radio Frequency Warner for the pipeline pattern (see Fig. 4.7), and Navigation System for the subservice pattern (see Fig. 4.9). The following example shows that the general pattern includes all cases that are not covered by the special case patterns, as it is composed of two input channels and one output channel, while the other output channel is not in use.

The $\mathcal{L}ightSystem$ automatically switches on the light when the sensor detects too little luminosity. The interaction may be overruled by the driver. The system is depicted in Fig. 5.9.

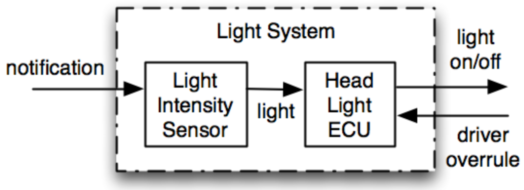


Figure 5.9: Decomposition of an Example from the Light System.

In this example, the interface consists of two input channels, one from the sensor, the *notification*, and one from the driver, the *driver overrule*. The output is a single communication channel, the one that switches the *light on/off*. The assumption/guarantee specification for $\mathcal{L}ightSystem$ is:

$$A_{LightSystem}(notification) \wedge A_{LightSystem}(overrule) \Rightarrow G_{LightSystem}(light)$$

The subsystems are the $\mathcal{L}ightIntensitySensorECU(LIS)$ and the $\mathcal{H}eadLightECU(HL)$. The assumptions and guarantees for the system and the subsystems are given as:

A_{LIS}: The light intensity information is available.

G_{LIS}: The system indicates whether light is necessary.

Subsystem Spec. LIS: $A_{LIS}(notification) \Rightarrow G_{LIS}(light)$

A_{HL}: The information about whether light is necessary is available, optionally, a driver overrule can occur.

G_{HL}: The system switches on the light if necessary.

Subsystem Spec. HL: $A_{HL}(light) \wedge A_{HL}(overrule) \Rightarrow G_{HL}(light\ on/off)$

Composition leads to:

$$A_{LIS}(notific.) \wedge A_{HL}(overrule) \wedge A_{HL}(light) \Rightarrow G_{LIS}(light) \wedge G_{HL}(light\ on/off)$$

Reduction: $G_{LIS}(\mathit{light}) \Rightarrow A_{HL}(\mathit{light})$

Result: $A_{LIS}(\mathit{notification}) \wedge A_{HL}(\mathit{overrule}) \Rightarrow G_{HL}(\mathit{light\ on/off})$

The verification that the assumptions and guarantees can be composed such that the decomposition is proved is not a mandatory step within DeSyRe. The patterns work, a subsequent composition and verification show whether correct assumptions and guarantees for the subsystems have been chosen. Therefore, the verification can be performed as optional but recommended step for quality assurance.

5.5.3 Design

The artifacts for the content category design on the logical level differ more from the artifacts on the usage level than the other content categories. For that reason, they have already been defined and described in Sec. 2.5.3.

The requirements engineer develops the artifacts from the design artifacts on the system level and the context and requirements artifacts on the subsystem level. The service graph of the design category on the system level (see Sec. 2.5.3) served as basis for the system decomposition as described in Sec. 5.4.2. The *Rationale* captures the design decisions that have been taken for the decomposition of the system. A *Component Model* (Sec. 2.5.3) is the documentation of the structural view on the decomposition and can be refined hierarchically. A *Behavior Model* (Sec. 2.5.3) is the system view on the interactions defined in the use case scenarios on the logical level and accounts for the constraints documented in the *Operational Environment*.

Thereby, each component represents a subsystem. Consequently, the requirements engineer deduces a service graph and an interface description for each of the subsystems by using the use cases and the operational environment descriptions of the context and requirements artifacts of the respective subsystem specifications. Later on, these new black box specifications of the subsystems will be turned into white box specifications by the respective subcontractors who realize the subsystems.

Running Example. The service graph of the driver assistance systems was already depicted in Fig. 5.7. The service graph that is deduced from the radio frequency warning system from use case scenarios as presented in Sec. 5.5.2 is depicted in Fig. 5.10.

5.5.4 Compositionality

When decomposing a system into subsystems, even more with the intention of developing those subsystems distributedly, compositionality is an important issue. Janssen [Jan97] observed that a compositional approach enables the developer to think of the system as a composite set of behaviors, which means that he or she can factorize design problem into smaller problems which can then be handled one by one.[3]

Apart from the intuitive understanding of compositionality, how can compositionality be captured formally? Concerning language in general,

[3]Janssen further concluded that “thus compositionality forms a reformulation of old wisdom, attributed to Philippus of Macedonia: divide et impera” [Jan97, p. 1]

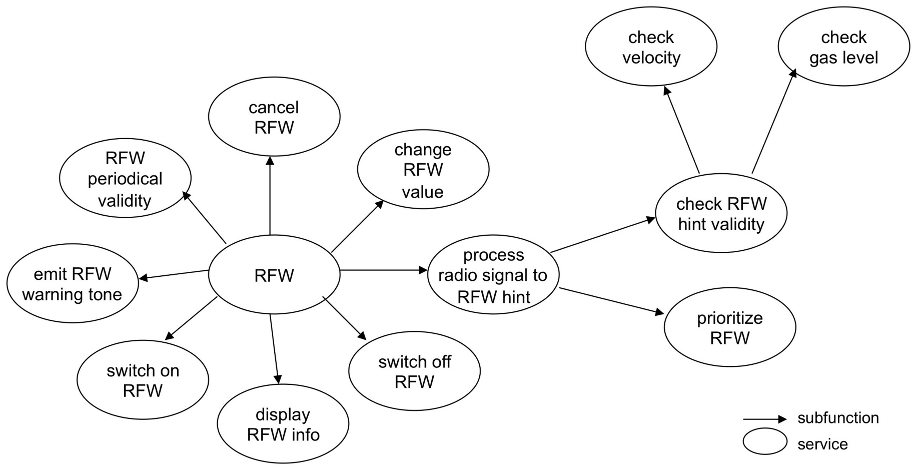


Figure 5.10: Service Graph Overview of the RFW

Montague [Mon70] suggests to capture the principle of compositionality formally through a homomorphism between the expressions of a language and the meanings of those expressions. However, a discussion of an encompassing formalization of language with a syntactic algebra for application in requirements engineering would go too far for the purpose of DeSyRe. Therefore the method is limited to aiming for compositionality without such a complete formalization and refers to, for example, Fleischmann [Fle08] for further reading on that topic related to the domain of requirements engineering for embedded systems.

A means for tackling compositionality is to master its strongest opponent, according to the Stanford Encyclopedia of Philosophy [Sza07], the consideration of the context. The same specification can lead to crucially diverging results if the context is not defined appropriately and meaningfully. This reinforces putting emphasis on the explicit documentation of the context within the artifact model, not only on the usage level but also on the lower abstraction levels.

Compositionality can only be actually guaranteed when using a formal approach and verification techniques. An example for the realization of guaranteed compositionality is the verification of operational semantics as described, e.g., by Larsen [LL91], but this is out of scope for the work at hand.

Within DeSyRe, compositionality is given for the usage of the requirements decomposition patterns, as shown in Sec. 4.6.

Running Example. For the driver assistance systems, the compositionality of the requirements that were refined using assumption/guarantee specifications is verified by (re-)composing the subsystem specifications. For other derived artifacts, additional quality assurance is required to verify the compositionality.

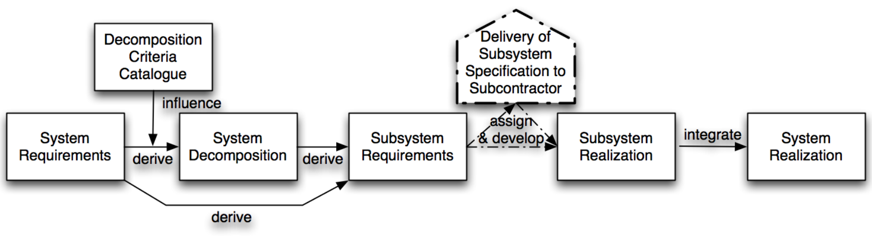


Figure 5.11: Delivery of Subsystem Specification in Process DeSyRe

5.6 Delivery of Subsystem Specification

At this point of the process, the systematic refinement of the artifacts from the usage level lead to a self-contained specification for the subsystems that can be used independently. Subsequently, the subsystem artifacts can be put in one document and handed on to the organizational unit responsible for the development of the subsystem, whether in-house or at a supplier's site.

At this stage, the DeSyRe process is suspended until the integration and the gap is filled by the standard implementation and testing phases of the developing organizational unit.

After the implementation, the developed subsystem has to be integrated with the other developed subsystems and into the overall system.

Integration is a phase that currently still challenges system developers (or original equipment manufacturers). First, this is because of distributed development and therefore difficult communication, as for example observed by Herbsleb and Grinter [HG99].

Second, high effort in integration is mostly due to insufficient specification of subsystems. How to improve the latter and instead develop a consistent, self-contained subsystem specification was described in the past sections, thereby laying the most important foundation for smooth integration.

The next section deals on how to integrate and / or reuse a subsystem and its documentation. As the DeSyRe method description does not include any tasks with regard to the actual implementation of the system, it is assumed at this point, that the realization of the subsystems has been accomplished by the separate development units (suppliers and organizational units of the contractor). Therefore, the next stage of the general software systems development process is the integration phase.

Running Example. For the driver assistance subsystems, all artifacts concerning the radio frequency warning system are put together as subsystem specification and are delivered to the subcontractor. All artifacts concerning the adaptive cruise control are gathered in another subsystem specification and are delivered to another subcontractor for realization.

5.7 Integration and/or Reuse

Finally, the last step of the development process is the integration of the realized subsystems (Fig. 5.12).

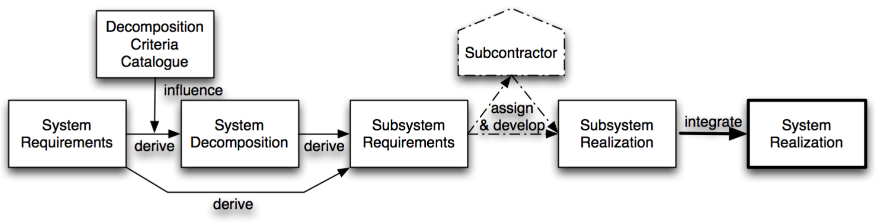


Figure 5.12: Integration and/or Reuse in Process DeSyRe

Integration or re-integration denotes the integration of a developed subsystem into the overall system whose specification was the point of origin for the extraction of the subsystem specification.

For reuse, there are different situations: within a product line, in a component-off-the-shelf approach, or simply ad-hoc when developing a new system that resembles an older one which has already been developed.

5.7.1 Integration

In case of integration of a subsystem whose specification was directly derived from the system specification with DeSyRe, the integration should not reveal misalignments between the system specification and the subsystem specification in case the system specification is correct. Therefore, any potentially arising compatibility problem should already have been detected during module test on the subsystem level when the complete subsystem is validated against its specification. Else, the initial overall system specification may have contained errors from the beginning.

Running Example. For the driver assistance systems, integration of the developed subsystems radio frequency warning system and adaptive cruise control should not reveal any problems at that stage. Furthermore, the subsystem specifications are reusable if updated by using a continuous change management process.

5.7.2 Reuse

In case of reuse, the steps to be taken are to identify an appropriate subsystem and check its compatibility. In case no adequate subsystem can be found, it is necessary to analyze and integrate additional requirements and reiterate development.

Subsystem Identification. The following steps only apply in the case when the subsystem is not yet chosen or has not been developed especially for the surrounding system under development. For that case it is necessary to identify appropriate subsystems, by a search which can be accomplished according to their interface specification.

The prerequisite for such a search is an existing knowledge base that stores all the interface specifications and allows for queries. Imagining such a database is already set up and filled to a useful degree, a string search for keywords about the desired functionality in the fields *Name* and *Purpose* should find the potentially adequate subsystems. These candidate subsystems subsequently have to be checked for their actual compatibility to the surrounding system under development.

Compatibility Check. The most important demand for compatibility is that the interfaces and borders between subsystems are cleanly and consistently defined with all constraints.

Garlan et al. have discussed that architectural mismatch stems from mismatched assumptions a reusable part makes about the system structure it is to be part of. They blame this problem on conflicts of these assumptions with the assumptions of parts of the new surrounding system, which are almost always implicit, thus they are extremely difficult to analyze before building the system [GAO95].

Therefore, the appropriate modeling of the subsystem borders and their context is crucial to avoid mismatches when integrating the subsystem into a (new) surrounding system. The guiding question is: What information can we use as is and what do we have to add? For the information that is already present we have to decide whether the given form is already appropriate or if we have to adapt a form to avoid dragging along to much information.

The constraints can roughly be divided into hardware constraints (e.g., physical limits) and software constraints (e.g., data types) and each of the types can be categorized as static (e.g., resources) or dynamic (e.g., timing). For ensuring compatibility, it is necessary to check all types of constraints that are captured in the artifacts Operational Environment, Interface Specification, Data Model, and Behavior Model.

Analysis and Integration of Additional Requirements. In case that there was no exactly matching subsystem found during the Subsystem Identification, the solution would be to perform an analysis of the additional requirements that could not be satisfied by the subsystem so far.

Analyzing the additional constraints and requirements provides an answer to the question what would have to be added to the subsystem, or what would have to be changed within the subsystem to fully meet the demands imposed by the surrounding system under consideration. Subsequently, when those efforts have been named, a cost calculation has to be performed to determine whether the adaptation is worth the efforts.

Finally, when that decision was made, the last step is to integrate the new requirements into the artifacts and check their consistency, then go back to the implementation phase for realizing the changes in the actual software and afterwards again return to ensuring the compatibility.

5.8 Implications

The approach brings a number of benefits, but it also imposes limitations. Both aspects are discussed in the following.

5.8.1 Benefits

This section presents the benefits of the application of the approach with respect to requirements engineering, architecture development, and reuse. The most specific benefits of this work are improved structuredness of the development of subsystem requirements and completeness of subsystem requirements specifications.

Structured Derivation of Subsystem Requirements. The proposed approach is the first published guide for systematically deriving *sub*system requirements from system requirements. It provides a detailed description of the steps the requirements engineer and the system designer have to perform. The requirements engineer elicits the system requirements, the system designer performs the initial system decomposition using the catalogue, and the requirements engineer derives the subsystem requirements using the patterns.

Completeness of Subsystem Specifications. With the process steps for derivation of subsystem requirements, all necessary information from the system requirements specification is transformed into the appropriate form for the subsystem requirements specification that is to be delivered to a subcontractor. The term *completeness* is to be seen relative to what is considered to be necessary information for successful system development. Naturally, the precondition for completeness of the subsystem requirements is completeness of the system requirements.

Benefits of Model-based Requirements Engineering. In the situation in embedded systems described in the work at hand, model-based requirements engineering (RE) can help to overcome some of the difficulties related to natural language requirements [PBKS07]. In model-based RE, conceptual models such as goal models, scenario models, or function models are used to specify requirements. Model-based RE offers several advantages: A model represents a well-defined view of the planned system. Design constraints and design choices can be documented in separate models, e.g., using an architecture description language. Models can be automatically checked, e.g., for consistency between different views or to identify the absence of certain requirements during tool-supported simulations. Models support communication among the stakeholders and achieve a common understanding of the planned system.

As the approach at hand works artifact-based, it features all the advantages of model-based requirements engineering.

Benefits of Architecture Documentation. As discussed by Beneken, architecture and its appropriate documentation are fundamental for successful project management in big software development projects [Ben08, Chap. 1]. The DeSyRe approach supports explicit definition of the first design sketch and documentation of the architecture by providing the decomposition criteria catalogue (Chap. 3) and the guiding process (Chap. 5) for application of the catalogue and documentation in artifacts (Chap. 2.5.3). The artifact model and the decomposition guidance offer a systematic approach to information gathering and architecture development. In the content category "design" of the artifact model, the views of behavior and interfaces are already captured.

Benefit of Systematic Reuse. The DeSyRe approach also provides support for reuse: in form of specification artifacts that compose a modular artifact model (Chap. 2.5.3), which adheres the OMG Reusable asset Specification [OMG04], and with a short description of which steps to perform for the reuse of a subsystem specification in Sec. 5.7. Furthermore, reuse is also one of the criteria in the decomposition criteria catalogue (Chap. 3).

As reuse has been proven to lower maintenance load and improve software quality by Stützle [Stü02, p. 108], the approach helps to achieve these goals by supporting reuse.

5.8.2 Limitations

The potential limitations of the approach at hand are:

1. the transferability to different surrounding development processes,
2. and the required expertise or competence for using the approach.

Transferability to Surrounding Processes

Another limitation could be that it requires effort to integrate it into surrounding, established development processes or, even more desirable with respect to facilitated integration, development process models. In case there is an established process like RUP [Kru00], V-Modell XT [Bun08b] or others, the question is whether it requires much effort to include the approach at hand into that framework.

Implications. Additional effort for adaptation of existing processes reduces the probability of application in practice. It is likely that the requirements documentation differs somewhat from the documentation used in the approach at hand, in explicit there may not even be an established artifact model.

Counter Measures. Neither the surrounding process nor the type of requirements documentation influence the applicability of the approach. The guidance on system decomposition can be used independently at the appropriate stage of the development process. The information sources for the decomposition criteria are described such that they can be found in any information model used for the initial requirements documentation. The guidance on requirements decomposition and refinement can be applied to whichever form of requirements documentation is established, whether it be natural language text or a different artifact model.

Required Expertise

A final potential limitation is the required expertise or competence that is required to perform the approach at hand. A developer with limited background knowledge on requirements engineering will probably have difficulties with the application.

Implications. The higher the effort for learning how to use the approach, the less is the motivation of the developers and therefore the probability for its application in practice. Tutorials have to be held in seminars or provided as documents for individual learning and reference.

Counter Measures. The estimation is that the approach at hand can be taught in approximately two hours: One hour presentation and explanation and another hour for exercises and their discussion. Little experienced developers may experience difficulties if their general knowledge of requirements engineering is low, but not specifically related to the approach at hand.

Summary. This chapter presented the guiding process for the DeSyRe approach to get to a complete subsystem requirements specification from the system requirements. It integrates the concepts of the decomposition criteria catalogue and the requirements refinement and applies them using a requirements engineering artifact model. The steps are illustrated with examples from the driver assistance systems, which are described in more detail in the next chapter.

Chapter 6

Evaluation and Assessment

Contents

This chapter presents the evaluation of DeSyRe in case studies, experiences gained in the case studies, and a small survey evaluation with respect to the usefulness of the approach for practitioners.

There are two research objectives for the evaluation of the method: applicability and usefulness. The applicability is evaluated with a case study performed for the driver assistance systems, reported on in Sec. 6.1. The usefulness of the approach was evaluated by a questionnaire handed out to practitioners after a tutorial on the approach, reported on in Sec. 6.2.

6.1 Case Study on Applicability

In the following, the case study is presented and discussed inspired by the structure recommended by Runeson and Höst [RH09].

6.1.1 Research Objective

The research question for the case study is defined according to the goal definition template by Wohlin et al. [WRH00]:

Analyze the DeSyRe approach
for the purpose of evaluation
with respect to the applicability
from the point of view of the requirements engineer
in the context of driver assistance systems.

6.1.2 Study Object

The driver assistance systems were chosen as case study for the purpose of validating the applicability of the approach presented in the work at hand. As they provide a wide range of different granularities and types of requirements with a sufficient degree of detail, they qualified as suitable for an evaluation of DeSyRe. The objective was to perform the complete process described in Chap. 5.

6.1.3 Study Design

The case study was performed as action research by the author. She applied the complete DeSyRe method to design and document the decomposition of driver assistance systems and derive the requirements for the subsystems radio frequency warner and adaptive cruise control. These two subsystems were already introduced shortly in Sec. 2.6. Thereby, she began with the starting point in Sec. 5.3, followed by deducing the system decomposition (Sec. 5.4), performing requirements refinement (Sec. 5.5), and documenting the respective contents with help of the artifact model.

The initial input for the case study was composed by a number of information sources: General background knowledge about driver assistance systems was gained during research projects with BMW, Daimler, Bosch, and Siemens as well as through literature research, e.g., [Sta09, Eur06, LCJZ08, WKV08, BV06, BDS08, LS04]. The original source document of the requirements specification of the RFW system by Daimler AG is available in German only [Ris07]. The original requirements that have been used in the ACC case study are documented in [FFH$^+$09b].

6.1.4 Execution and Results

For easier reading, the contents and results of the case study are set in *italic sans serif* while the supplementary explanations are set in normal font. All artifacts have been developed from scratch on the basis of common knowledge on driver assistance systems and the cited documents [Ris07, FFH$^+$09b] to avoid confidentiality issues.

System Requirements

The following passage lists requirements that apply to the entirety of DASs, captured in the artifacts of the usage level as defined in Sec. 2.5.3. As the usage level describes the system in a black-box manner, these artifacts (see Fig. 5.3) can all be developed before analyzing the decomposition. They serve as information basis for the decomposition.

Context

The artifacts for the context on the system level are the system vision, the goals (business goals and quality goals), the stakeholders, and the operational environment.

System Vision. The system vision is an image of the system to be developed: "A well-interrelated network of driver assistance systems shall support the driver in demanding traffic situations and thereby increase the safety of the vehicle."

Stakeholders. *Stakeholders are persons and institutions that are affected by the development and/or the operations of a system.*

1. *Driver, may be accompanied by other passengers.*
2. *Original equipment manufacturer, can be further detailed in marketing person, system architect, software engineer, quality assurance, lawyer.*
3. *Supplier, can be further detailed, like the OEM, but in this case is only regarded as one stakeholder.*
4. *Customer engineer, for problem reports and upgrades.*

Goals. *The business goals sum up the economic and customer-related aims of the system:*

1. *Improve market competition.*
2. *Safeguard investment, the vehicle.*
3. *Provide active safety.*
4. *Fortify trust of driver in brand.*

The quality goals determine specific characteristics that the system shall demonstrate apart from the actual functionality:

1. *High reliability and robustness.*
2. *Maximize utilisation of ressources.*
3. *Improve the communication between the different DAS.*

Operational Environment. *An illustration of the system environment is provided in Fig. 6.1. It depicts the system with its surrounding related systems and respective interacting stakeholders to provide an overview of the operational environment.*

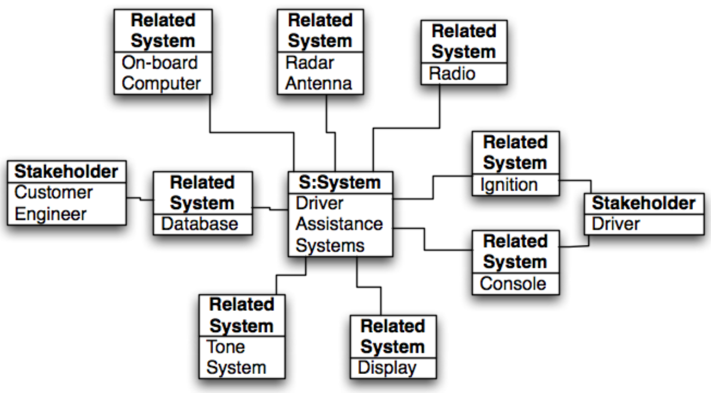

Related System On-board Computer
Related System Radar Antenna
Related System Radio
Related System Ignition
Stakeholder Customer Engineer
Related System Database
S:System Driver Assistance Systems
Stakeholder Driver
Related System Console
Related System Tone System
Related System Display

Figure 6.1: Operational Environment of DAS.

Requirements

The following use cases were gathered in a brainstorming session with BMW employees after identifying the system vision and the goals. Not all of the following use cases are realized by ACC or RFW.

Use Cases for DAS

1. *The driver is about to fall asleep. The system omits a warning tone.*
2. *The driver is about to change lanes. The system displays a vehicle in the blind spot.*
3. *The driver is using cruise control and approaches a vehicle ahead. The system decelerates.*
4. *The driver is driving 100km/h although there is a speed limit of 80km/h. The system displays a warning.*
5. *The driver is distracted by his kid on the backseat. When turning around to face the kid while talking, he unintentionally departs from the lane. The system omits a warning tone and displays a designated "lane departure" symbol.*
6. *A jogger in dark clothes is running on the shoulder of the road at night. The system notifies the driver by displaying a designated "human in way" symbol.*
7. *The driver is using cruise control and is about to overtake another vehicle. He sets the indicator to change lanes. The vehicle accelerates.*

The following functional requirements are not assigned to one specific use case but apply for all DAS.

Functional Requirements for DAS

1. *The vehicle shall keep a certain clearance distance to preceding vehicles, depending on the current speed and driving situation.*
2. *The driver has the final authority, not the vehicle.*
3. *The displays may not restrict the sight of the driver.*
4. *The driver may not be overly distracted by the DAS.*
5. *The DAS may not charge the busload more than xxx kBits/s.*
6. *All DAS shall be deployed onto one ECU.*

They are later on decomposed and refined into subsystem requirements for ACC and RFW.

Design

Service Graph. The service graph in Fig. 6.2 displays the system services. In current practice, there is hardly any usage-related interaction between driver assistance systems. Prioritization of warnings is an important issue for not stressing the driver by too much information at the same time, but this is a quality requirements which is not explicitly listed within the services that the systems provide.

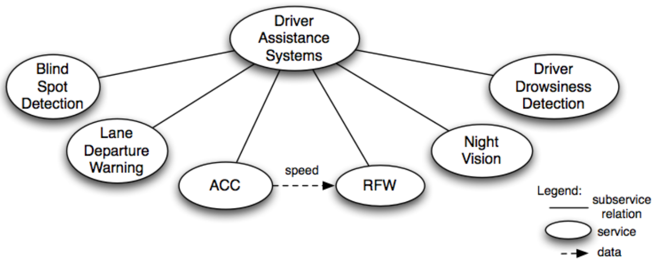
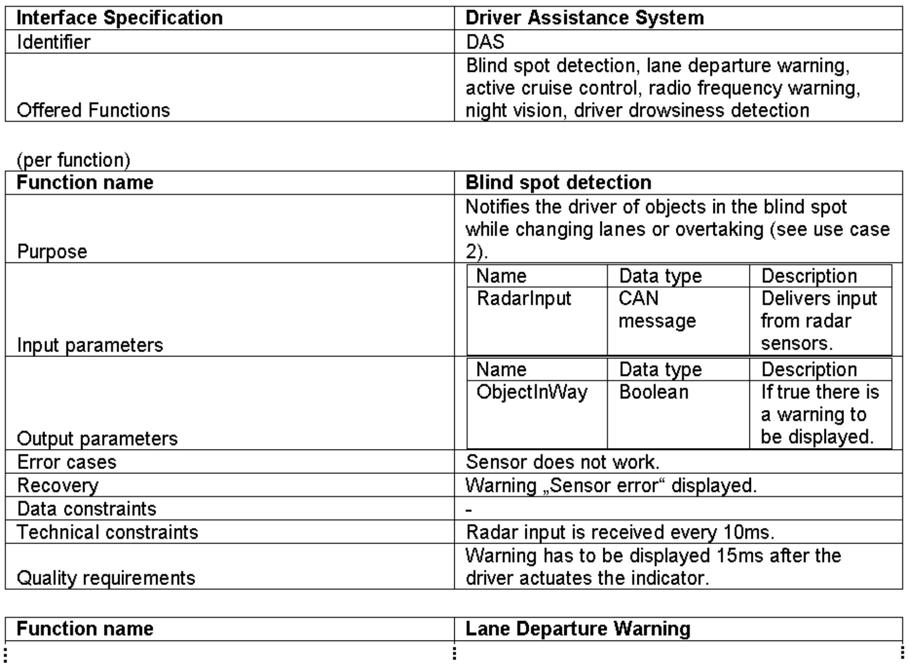


Figure 6.2: Service Graph of DAS.

Interface Specification. Fig. 6.3 displays part of the interface specification of the driver assistance system. The header lists the functions that are offered by the system and, subsequently, the interface is specified per function. In this case, only the function blind spot detection *is further detailed with parameters, error cases, recovery, and quality requirements.*



Interface Specification	**Driver Assistance System**
Identifier	DAS
Offered Functions	Blind spot detection, lane departure warning, active cruise control, radio frequency warning, night vision, driver drowsiness detection

(per function)

Function name	**Blind spot detection**		
Purpose	Notifies the driver of objects in the blind spot while changing lanes or overtaking (see use case 2).		
	Name	Data type	Description
Input parameters	RadarInput	CAN message	Delivers input from radar sensors.
	Name	Data type	Description
Output parameters	ObjectInWay	Boolean	If true there is a warning to be displayed.
Error cases	Sensor does not work.		
Recovery	Warning „Sensor error“ displayed.		
Data constraints	-		
Technical constraints	Radar input is received every 10ms.		
Quality requirements	Warning has to be displayed 15ms after the driver actuates the indicator.		

Function name	**Lane Departure Warning**
⋮	⋮

Figure 6.3: Example Interface Specification.

Decomposition of the System

At this point, all artifacts that serve as information basis for the decomposition of the overall system (see Sec. 5.3) have been presented. This section describes the application of the decomposition steps defined in Sec. 5.4.

Consideration of the Catalogue

When used for the first time in a company, the catalogue may be reviewed and probably customized to certain business strategies of the company (see Sec. 5.4.1). This would be accomplished by stakeholders from the management. Then the users of the catalogue, the requirements engineers, make themselves familiar with the catalogue and prioritize the optimization factors described in Sec. 3.2.1 according to the business goals that are documented in the artifact *Goals*.

Goals and Optimization Factors. *Naturally, most business goals cannot be mapped one to one to an optimization factor, because the optimization factors are related to systems engineering while the business goals stem from marketing and management. Instead, the business goals can be supported through optimizing the system in an adequate way. For example, the market competition can be supported through building a reliable system.*

A more direct relation is observable between the quality goals of the "Goals" artifact and the optimization criteria. "High reliability" (QG 1) can be mapped to reliability, "Maximize utilization of resources" is a cost factor, and to improve the communication between the different DASs requires controllability.

Goals and Decomposition Criteria. *The tables Tab. 6.1, Tab. 6.2, Tab. 6.3, and Tab. 6.4 list and analyze the decomposition criteria from the reference catalogue (Sec. 3.3 to Sec. 3.6) for the driver assistance systems.*

Table 6.1: Directive Decomposition Criteria in DAS

Organization	Infrastructure	Conway's Law says that the system will mirror the developing organization's structure, but at the same time, developing organizations will also be structured according to the systems they develop. The departments at BMW are already organized according to the different driver assistance systems.
	Business Rules	No explicit business rules are known of, that would influence the decomposition of DAS.
	Experience	The experience of the developers is about ten years in average, their background is mainly mechanical and electrical engineering.
Legislation	Laws	German Road Traffic Act "StVO" [Bun07] , German Road Traffic Admission Act "StVZO" [Bun08c], Electro-magnetic Compatibility Act "EMV" [Bun08d] and others, but none of them explicitly influences the decomposition.
	Standards	Different German Industry Norms "DIN" apply, but none of them is relevant for the decomposition.
Economics	Reuse	All of the DAS have already been developed in the previous vehicle series, so reuse is strongly expected.
	Cost Models	Cost of the hardware parts is crucial as even minor amounts of money sum up for the number of produced vehicles. Software development costs are considered less critical, but in this case are connected with reuse, as a decomposition for high reusability will decrease the costs for the next iteration.

Table 6.2: Functional Decomposition Criteria in DAS

Usage Services	*The usage services that are present in the use cases in Sec. 6.1.4 can be grouped according to their functionality. This is the most intuitive decomposition.*
Functional Dependencies	*For the use cases in Sec. 6.1.4 there are some functional dependencies. For example, use case 2 is also relevant for use case 5 in case of a vehicle in the blind spot when departing the lane.*
Unwanted Feature Interaction	*Interaction occurs in case of two DAS responding at the same time to a traffic situation. For example, the cruise control accelerates when the driver sets the indicator to pass another vehicle but at the same time the system detects a vehicle in the blind spot and therefore has to stop the acceleration to prevent a crash with the vehicle in front as the driver cannot change lanes. Therefore, the decomposition has to allow for a resolver for the different DAS.*

Table 6.3: Quality Decomposition Criteria in DAS

Functionality	*In the definition of the ISO 9126-1 [Int01], functionality includes suitability, accuracy, interoperability, compliance, and security. Naturally, these factors are important for DAS but no explicit impact on the decomposition of DAS could be determined.*
Reliability	*Maturity, recoverability, and fault tolerance are crucial for all vehicles participating in road traffic. For system decomposition, this can mean that certain subsystems are grouped onto redundant electronic control units (to prevent harm in case of malfunction).*
Efficiency	*Performance is required in terms of behavior, as for any real time system, and performance in terms of resources, to keep the size and costs of the hardware low.*
Maintainability	*Changeability and testability are important for reusability. This leads to a typical modular decomposition with high cohesion and loose coupling.*

With prioritized optimization factors and a reviewed criteria catalogue, the requirements engineer is set up for analyzing dependencies between potential subsystems.

Dependency Analysis for Driver Assistance Systems

Before the decomposition of the system can be conducted, the dependencies between the potential subsystems have to be analyzed. Concrete examples for interaction between driver assistance systems are:

- *ACC and navigation: Interconnection between ACC and the navigation system. The navigation data is used to calculate the curviness of the roadway ahead and to adjust dynamic parameters to it.*

- *Braking and ACC: Conjoint use of the far range radar for the intelligent braking system and the ACC. The two systems also align their warning algorithms for system boundaries to prevent unwanted double warnings in the case of both systems being active.*

- *Driver Surveillance System: usage of multiple input parameters (mainly operator control actions of the driver) to calculate the driver's attention and adjusts the warning thresholds.*

- *Interferences of DAS: Two DAS are active at the same time and one has to overrule the other in certain circumstances, for example the cruise control*

Table 6.4: Technical Decomposition Criteria in DAS

Communication Requirements	Adequate communication has to be supported for real time interaction between the different DAS and with the rest of the vehicle.
Technical Constraints	Hardware topology and resources imply a number of technical constraints that are important for the final deployment onto the technical architecture but not for the decomposition with respect to the logical architecture design. Furthermore, general potential bottlenecks arise from shared sensors where the input is used by more than one system as well as shared output channels that are used by more than one system, for example the head unit display.
Legacy Systems	There is the surrounding system environment to be taken into account with technical details as specified in the respective interface descriptions.

could be overruled when a future "fog assistant" senses approaching heavy fog.

These dependencies were not documented explicitly within any source documents, but are domain knowledge from systems engineers at BMW that was gathered during informal interviews.

Division into Subsystems

Recalling the described alternatives in Sec. 5.4.2, one possible conclusion is to apply a decomposition primarily according to the identified services, the user-perceived functionality. One usage service cluster, one logical function group, one electronic controller unit. It is not optimal with respect to resources and space capacity within the vehicle. A different approach might improve the utilization of resources but offer less flexibility for the design of the individual usage services.

In traditional development in practice in the automotive domain, the abstraction level of the logical architecture is often skipped. In the approach applied of the work at hand, the logical architecture and the technical architecture are designed separately.

For the technical architecture and the deployment, some of the decision factors for sharing an ECU are the input and output channel locations and potentially shared logic. In detail, the criteria for partitioning the electronic/electrical system architecture in vehicles are, according to [Rei09]:

- *Homogeneity / heterogeneity of requirements (on the software-in-the-loop level, processing requirements, resources, availability, ...)*
- *Communication requirements*
- *Functional interrelation (related functions integrated or in spatial proximity, complex functions not widely distributed, ...)*
- *Geometric criteria (space, wiring length, connectors, power lines, ...)*
- *Environmental conditions (humidity, temperature, electromagnetic compatibility, ...)*
- *Communality, compatibility*
- *Costs (integration level, take rate, new design, variant, ...)*
- *Service (accessibility, diagnosis, costs, ...)*
- *Scalability, extendability over architecture life cycle*

- *Variability (basic and high versions of vehicle)*

After deciding the initial system decomposition, the requirements specifications for the subsystems ACC and RFW were elaborated. This step takes the system requirements and the system decomposition into account and systematically refines the requirements for the subsystems.

Subsystem ACC

This section presents the developed and documented requirements of the subsystem "Active Cruise Control".

Context

System Vision

> *The driver assistance system* Active Cruise Control *(ACC) is an intelligent speed control system that automatically maintains a pre-defined distance to the vehicle in front.*

The system vision for a subsystem can in this case not be derived from the system vision of the surrounding system, because the idea of the driver assistance systems was phrased too abstractly to deduce any subsystems. It is the idea of the system to be developed agreed upon by all stakeholders, which has to be imagined creatively.

Goals. Some of the DAS goals can be refined for the ACC subsystem. For example, the goal Provide active safety *can be concretized into* Provide active safety by preventing rear-end collisions. *Other goals can not be refined in such a way that the additional information would prevent benefits, and therefore remain unchanged, for example the goal* Safeguard investment. *Additionally, there may be new goals that stakeholders require for the subsystem that were not yet present on the system level, for example, "Achieve faster adaptation of speed than market competitors.".*

Stakeholders. The stakeholders are the same as for the complete DAS: Driver, OEM, supplier, and customer engineer.

Operational Environment. The operational environment of the ACC subsystem in Fig. 6.4 tailors the DAS operational environment in Fig. 6.1. The related systems Radio *and* Tone System *are not present any more as they are not relevant for the ACC. Instead, the* Breaking System *has been added as interaction possibility for the driver.*

ACC Requirements

The following requirements are the refinements from the overall DAS functional requirements listed on p. 128.

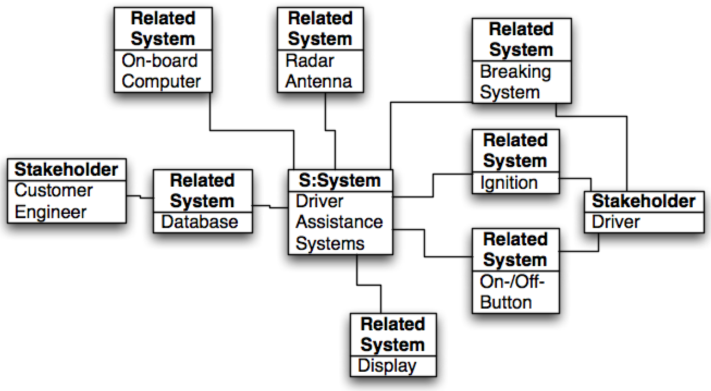

Related System On-board Computer
Related System Radar Antenna
Related System Breaking System
Stakeholder Customer Engineer
Related System Database
S:System Driver Assistance Systems
Related System Ignition
Stakeholder Driver
Related System On-/Off-Button
Related System Display

Figure 6.4: Operational Environment ACC

1. *Within the speed ranges of operation of the ACC system, the vehicle shall keep a clearance distance of one meter per km/h to preceding vehicles.*
   The clearance distance is refined for the usage service of the ACC.
2. *The driver can overrule the ACC system at any time by either braking, accelerating, or switching it off.*
   Thereby, the final authority is refined.
3. *The ACC displays or head up display may not restrict the sight of the driver.*
   The relevant displays are named.
4. *The driver may not be overly distracted by interference of the ACC, acceleration or braking shall occur smoothly.*
   Potential distraction by the ACC is named.
5. *The ACC may not charge the busload more than yyy kBits/s.*
   In this case, a model for calculating the busload is needed, as described in Sec. 4.7.2, Eq. 4.9.
6. *The ACC shall be deployed onto the same ECU as the other DAS.*
   The design decision for which ECU to deploy on has not been taken yet, but the requirement is adopted.

Further functional requirements that refer exclusively to the ACC system are documented in [FFH+09b]. It is the driver assistance system for Active Cruise Control with Pre-Crash Safety (PCS).

There are 28 informal requirements in natural language, but they are not well written. The first four are listed here, the remaining requirements are documented in App. A.1.

1. *Select target vehicle. Probability is calculated by radar data. Conditions: within the defined range; high probability; most close range vehicle.*
2. *Select target vehicle. Reject Parking cars.*
3. *Follow-up control of ACC system starts in case of target vehicle exists. Distance between two cars is controlled with in the target. Target*

acceleration is decided on deviation of distance and relative speed. Target acceleration is conveyed from Drive assist ECU to fusion ECU. Fusion ECU provides request to engine and brake components.

4. *Follow-up control of ACC system starts in case of target vehicle exists. Distance between two cars is configurable depend on vehicle speed.*

For example, requirements 3 and 4 are highly redundant. This lack of quality suggests to perform the quality assurance step described in Sec. 5.3 and perform a dependency analysis for the requirements.

Multiple dependencies. If there is a dependency between two requirements, there often applies more than one type of dependency. For example, between requirement (1) and (2), there apply the dependency types *refinement*, *concept bundle*, *feature bundle*, *logical component*, and *architectural bundle*. This can be explained through "dependencies between the types of dependencies" as a requirements refinement will always entail some of the other dependencies.

Ambiguous type of dependency. It is not always clear, which type of dependency is the most suitable, but it does not seem to make sense to always assign all potential types of dependencies. For example, is the dependency between (1) and (2) a *refinement*, *concept bundle*, *feature bundle*, *logical component*, or *architectural bundle* - or is it necessary to consider all of them?

Requirements Dependencies. The majority of detected dependencies were refinements, *i.e., there were eight occurrences. Furthermore one* concept bundle, *three* feature bundles, *two* logical components, *and one* architectural bundle *could be identified. All of them are listed in App. A.2. There were no examples found for architectural restrictions, architectural alternatives, conflict negotiation, or architectural decisions. Missing examples for the last four dependency types are likely to be due to using only the original informal requirements of the ACC instead of more formally refined requirements. It is not crucial to mark all possible dependencies for the realization of the system, but it is important to find them as they give hints on potential requirements inconsistencies or conflicts.*

Requirements Conflicts. There were no conflicts found within the ACC requirements. This was checked by manually reviewing all dependencies. Therefore the development can proceed straight away to the design artifacts.

Design

The design on the system layer is documented in the artifacts service graph, behavior specification, and interface specification.

Service Graph

Fig. 6.5 shows the service graph of the ACC subsystem. The ACC is composed by the three subservices Activation/Deactivation, Regulation, *and* Input*.* Activation *enables the subservices of the* Regulation*. Some of the subservices exclude each other, because the velocity of the vehicle can only be either slower than 31km/h, between 31 and 180km/h, or faster than 180km/h.*

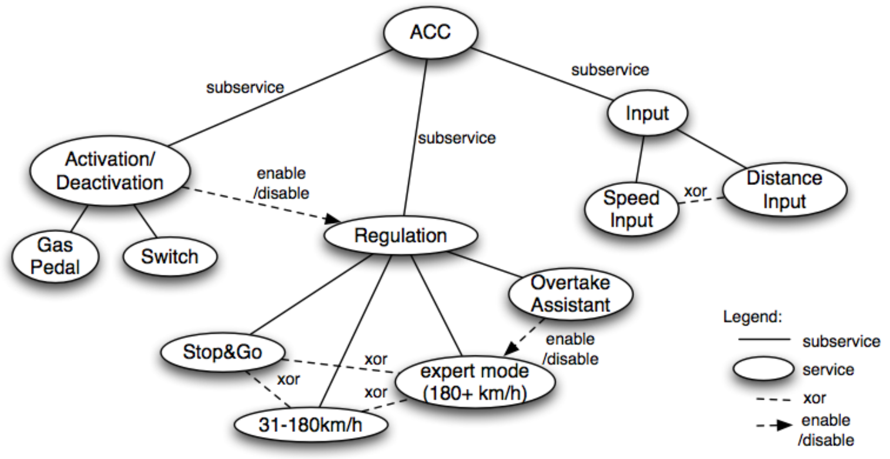


Figure 6.5: Service Graph of the subsystem ACC

Interface Specification

The interface specification is simply part of the interface specification that has been defined for the complete set of driver assistance systems, as the functionality of the ACC is already perceivable on the "outside" of the black box of driver assistance systems.

The preceding section captured the resulting requirements specification for the ACC on the usage level. For the case study, the work with respect to the subsystem ACC is complete. In the real development process, the subsequent step would lead to design on the logical architecture level.

Subsystem RFW

The requirements specification of the RFW system [Ris07] (in German) is the case study developed for the REMsES project [RDSS09]. The complete specification cannot be supplied within this document due to reasons of secrecy, but examples are taken in form of the artifacts presented in the following.

Context

System Vision.

> *The driver assistance system "Radio Frequency Warning" (RFW) supports the driver in coping with the information input from the surrounding environment by use of radio frequency signals. [Ris07, p. 6]*

As with the subsystem ACC, the System Vision for the RFW subsystem can not be derived systematically from the System Vision of the surrounding system. It is an aim of the system to be developed agreed upon by all stakeholders, which has to be imagined creatively.

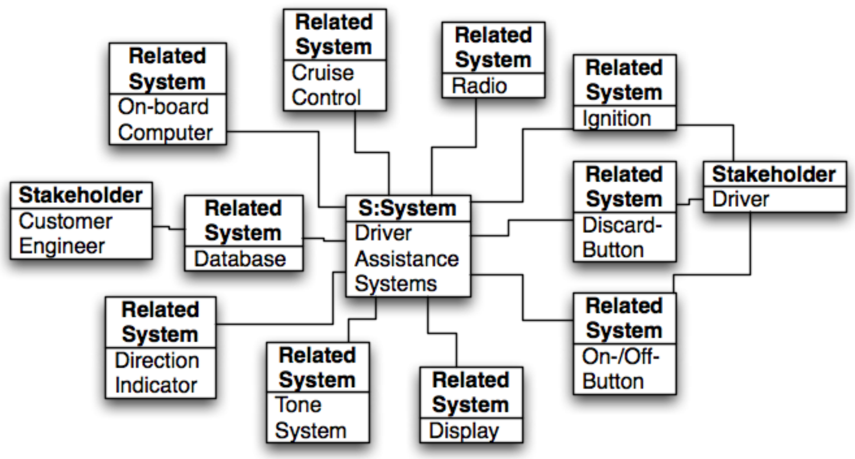


Figure 6.6: Operational Environment RFW

Goals. The goal "Provide active safety" can be refined into "Provide active safety by preventing that the driver misses a traffic sign which could cause an accident". Additionally, a new goal is: "Use different modalities within RFW to address the driver."

Stakeholders. The stakeholders are the same as for the complete DAS: Driver, OEM, supplier, and customer engineer.

Operational Environment. The operational environment of the RFW subsystem in Fig. 6.6 tailors the operational environment in Fig. 6.6. There is no Radar Antenna *any more, but there are the* Tone System*, the* Direction Indicator*, and the* Discard Button*.*

Requirements

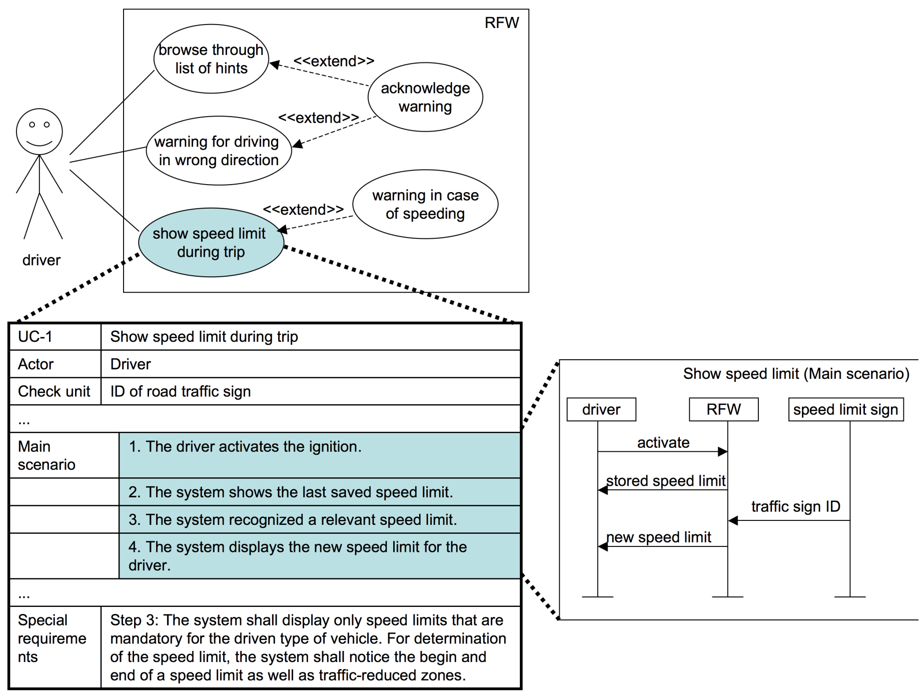
Common use cases of the RFW subsystem, for example browse through list of hints, warning for driving in wrong direction, *and* show speed limit during trip *and the detailed scenario for the latter use case are depicted in Fig. 6.7.*

Requirements Dependencies and Conflicts. There were examples for the same types of dependencies as for the requirements for the subsystem ACC, but no new insights gained. There were no conflicts found within the RFW requirements.

Requirements Refinement

The following requirements are the refinements from the overall DAS functional requirements listed on p. 128.

1. *Clearance distance* is not applicale to the RFW system, therefore the requirement is not refined.


RFW
browse through list of hints
<<extend>>
acknowledge warning
<<extend>>
warning for driving in wrong direction
warning in case of speeding
<<extend>>
show speed limit during trip
driver
Show speed limit (Main scenario)
driver
RFW
speed limit sign
activate
stored speed limit
traffic sign ID
new speed limit

UC-1	Show speed limit during trip
Actor	Driver
Check unit	ID of road traffic sign
...	
Main scenario	1. The driver activates the ignition.
	2. The system shows the last saved speed limit.
	3. The system recognized a relevant speed limit.
	4. The system displays the new speed limit for the driver.
...	
Special requireme nts	Step 3: The system shall display only speed limits that are mandatory for the driven type of vehicle. For determination of the speed limit, the system shall notice the begin and end of a speed limit as well as traffic-reduced zones.

Figure 6.7: Scenario RFW

2. *The RFW system can be switched off by the driver at any time.*
   As the RFW system does not actively interfere but only displays available information, the driver's authority can only be refined by the operation of switching off the system.
3. *The RFW display may not shine brighter than vvv lux in order to not restrict the sight of the driver.*
   The RFW does not use a head up display in this case, so it is sufficient to refine the requirement for the instrument cluster display.
4. *The warning tone of the RFW system shall be in an adequate volume to be noticed by the driver but not to startle them. The frequency of the displays shall not exceed the cognitive abilities of the driver.*
   This requirement refines and restrict the possible distractions by the RFW.
5. *The DAS may not charge the busload more than zzz kBits/s.*
   As well as in case of the ACC requirements, a model for calculating the busload is needed, as described in Sec. 4.7.2, Eq. 4.9.
6. *The RFW system shall be deployed onto the same ECU as the other DASs.*
   Again, the design decision for which ECU to deploy on has not been taken yet.

For the functional requirements of the specification of the RFW system, two specification documents have to be considered, namely the "system specification" [Ris07, p. 11ff] and the "component specification" [Ris07, p. 29ff] that gives further details. The following refinements can be made:

Interface Requirements. The "environment" requirements within the system specification [Ris07, Chap. 3] describe the interaction of the RFW system with the environment, for example: "The RFW system communicates with other systems via a CAN interface." [Ris07, p. 8] These interface requirements can be adopted for the component specification without changes. Therefore, the same requirement can be found in the component specification of the RFW system [Ris07, p. 27].

The same applies for the other requirements of the chapter, which all describe the interaction with the environment, i.e.:

- driving direction information [Ris07, p. 9],
- speedometer [Ris07, p. 9],
- on-board computer [Ris07, p. 10],
- display [Ris07, p. 10],
- tone system [Ris07, p. 10],
- velocity control system [Ris07, p. 10/11].

All of these are directly adopted for the subsystem as they impose interface constraints that are not further refined for subsystems.

System Requirements. The "system description" chapter within the system specification [Ris07, Chap. 4] encompasses functional and technical requirements.

The core functionality of the RFW controller is given in [Ris07, p. 12]: "The RFW controller is at the core of the RFW system. Here arrives the signal data of the RF reader and the rule system of the controller generates the respective output. For details, see component specification RFW controller Sec. 5.3.1 function description."

This is further detailed within the function requirements that are most relevant for the requirements refinement.

Function Requirements. The "technical requirements" chapter within the system specification [Ris07, Chap. 5] encompasses, inter alia, a description of the functionality of the RFW system. The description includes examples for requirements of the different patterns described in p. 64. There were occurrences of all requirements patterns, but with different frequency. The most frequent pattern is the response pattern. With a much lower frequency follow the absence pattern and, even with lower frequency, the precedence and existence patterns. These findings confirm the results of the survey by Dwyer et al. [DAC99]. Tab. 6.5 gives an example for each pattern. Tab. 6.6 shows the refinement of the requirement "Reading of Radio Signal Queue" [Ris07, p.36].

Table 6.5: Refinement of Functional Requirements of the RFW System

Ref.	Requirement	Pattern
[Ris07, p.31]	All radio signals are added to the queue. Exceptions are speed limits and clearings. These are stored separately as they nullify each other.	Existence
[Ris07, p.37]	The code shall be sent only after a code was sent from the queue.	Precedence
[Ris07, p.37]	If a discard-signal is received, the current entry is deleted.	Response
[Ris07, p.44]	The component may not produce short circuits.	Absence

Table 6.6: Refinement of Functional Requirement "Reading of Queue"[Ris07, p.36, RFW576]

Ref.	Requirement	Pattern
[Ris07, p.36]	If the queue is empty, go to branch "Maximum velocity".	Response
[Ris07, p.36]	Else send code symbol of the current entry to display as "SET_RASI_SMYBOL".	Response
[Ris07, p.36]	The function is called periodically and the symbol code is also sent periodically. For cycle time, see communication matrix.	Response
[Ris07, p.36]	In case of a warning, there is an additional warning tone message sent to the tone system.	Response
[Ris07, p.36]	Warning tones are emitted no more than once.	Absence

For the decomposition patterns described in Sec. 4.5.2, the following examples occur.

Subservice Decomposition Pattern.

- *Original Requirement: "The assigned priority of each incoming radio signal is looked up in the radio signal catalogue." [Ris07, p.36]*
  - *Assumption: There are incoming radio signals that are listed in the radio signal catalogue.*
  - *Guarantee: The priority of the radio signal is returned as assigned by the catalogue.*

The relevant subsystems are the RFW controller and the radio signal queue. The RFW controller uses the radio signal catalogue as a subservice to identify the priority of a radio signal. The requirement can be decomposed according to the subservice pattern (see Sec. 4.6.2) in the following way:

- *RFW controller: As the RFW controller is the only subsystem interacting with the interface, it has to satisfy the whole requirement "The assigned priority of each incoming radio signal is looked up in the radio signal catalogue."*
  - *Assumption: There are incoming radio signals that are listed in the radio signal catalogue.*
  - *Guarantee: The radio signal is assigned the priority given by the catalogue.*
- *Radio signal catalogue: "The priority of the radio signal is returned to the controller."*
  - *Assumption: A valid radio signal is received from the controller.*
  - *Guarantee: The appropriate priority is mapped to the received radio signal and sent to the controller.*

Pipeline Decomposition Pattern.

- *Original Requirement: "Each radio signal is listed in a radio signal queue ordered according to priority of the signals." [Ris07, p.36]*
  - *Assumption: There are incoming radio signals that have been assigned a priority by the radio signal catalogue.*
  - *Guarantee: The radio signal is placed at the slot corresponding to the priority within the queue.*

The relevant subsystems are the RFW controller, the radio signal queue, and the display. The three subsystems form a pipeline in order to fulfill the requirement. The requirement can be decomposed according to the pipeline pattern (see Sec. 4.6.1) in the following way:

- *RFW controller: "The system shall send the incoming radio signals with their priorities to the radio signal queue."*

- *Assumption: There are incoming radio signals that have been assigned a priority by the radio signal catalogue.*
- *Guarantee: The signal and its priority are sent to the radio signal queue.*

- *Radio signal queue: "The system shall represent the radio signal notifications according to their priority."*
  - *Assumption: The radio signal and its priority are received from the controller.*
  - *Guarantee: The radio signal is inserted into the queue at the slot corresponding to the priority so that the complete queue is ordered according to priorities.*

There are no examples for the general decomposition pattern within the RFW requirements as the system does not interfere with the vehicle control, but only displays available information for the driver.

Design

The first design sketch of the radio frequency warning system is depicted with the Service Graph. The functionality of the RFW system is shown in Fig. 6.8.

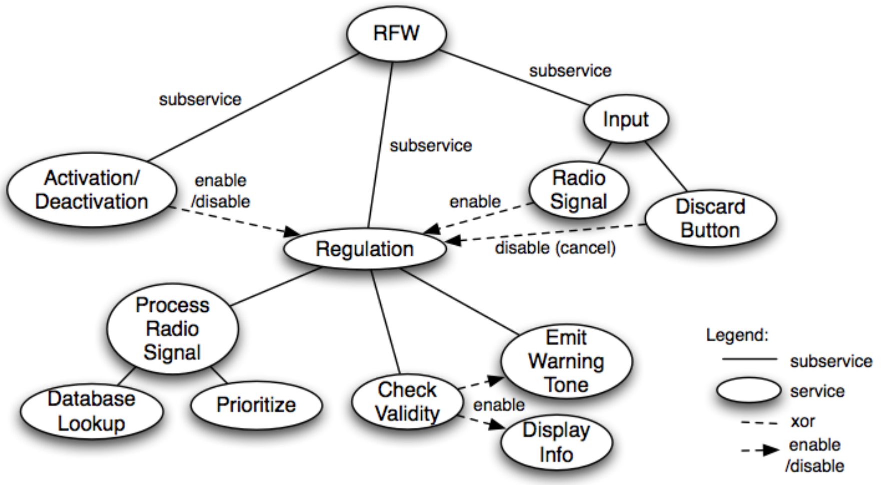


Figure 6.8: Service Graph of the RFW.

The overall system functionality RFW is decomposed and refined into subservices Activation/Deactivation, Input, and Regulation. Each of them is decomposed again into subservices. Some of them en- or disable others, for example, the Discard Button service cancels the current regulation.

6.1.5 Discussion

It was possible to develop all artifacts for the overall system specification of the system level and deduce the subsystem specifications for the two chosen subsystems. The process was applicable and did not reveal gaps.

Application of the Criteria Catalogue. The catalogue can be used as checklist, but in an application domain as well known and explored as the automotive domain, the logical decomposition tends to stay close to previous solutions. The question, when a logical architecture should be changed, can only be answered by comparing the old design and some alternative new designs through an architecture evaluation method. At the same time, it requires a trade-off between the effort of developing a new design and the impact of the shortcomings of the old design. On the technical architecture level, the system designer has to consider, that hardware development advances fast and hardware cost optimization is crucial for high production volumes.

Application of Requirements Refinement. Due to the limited amount of coarse-grained functional requirements, only a limited number of example requirements could be found that can be decomposed and refined to illustrate broader applicability. However, the smaller number of natural language functional requirements (in comparison to traditional natural language requirements specifications as still in practice in the automotive domain) is due to the graphical artifacts used for parts of the case study.

Requirements Decomposition Patterns. The question of which pattern to use depends on the point of view of a specific requirement. For example, for the requirements for the signal queue of the RFW system, the queue can be seen as pipeline between controller and display, or as subservice to either of them. This lies within the preferences of the requirements engineer who performs the decomposition. The important part is that no information gets lost.

Less requirements could be transformed by using the patterns than expected. The application of the patterns requires the possibility to represent them in assumption / guarantee style. This is the case for functional requirements that state details relying on perceivable inputs and outputs.

Application of the DeSyRe Guiding Process. The guiding process explains how to apply the concepts of the decomposition criteria catalogue and the requirements refinement in combination with an artifact model for requirements engineering. The process does not depend on that specific artifact model and may be substituted by a different one, which would require a more generic version of the process description. However, a more generic description was not desired to ensure applicability.

Feasibility of Requirements. Requirements that cannot be operationalized can only be copied, and therefore are hard to validate. This is the case for high-level requirements like goals that have not been transformed properly into system requirements. These requirements tend to stay vague because if the developer does not know how to satisfy them (e.g., by certain test measures), they cannot be realized. Consequently, they have to be refined sufficiently.

Appropriateness of the Artifact Model. The artifact model might have to be tailored to project-specific needs in order to support the requirements engineer and the system designer while at the same time not forcing them to produce more artifacts than necessary. Depending on the overall development process present in a company, it might be useful to adapt the artifact model, for example to better incorporate well-established tools.

Design Constraints of the Case Study. In general, the interaction between driver assistance systems is not very high. According to a BMW developer, this is due to the fact that these systems are usually extra equipment and the customer has to pay for each of them separately [Ebe09]. There is only little packaging, if at all. Therefore, the developer does not know for sure whether a certain driver assistance system that is interconnected with another driver assistance system is really present or not. It depends on the vehicle's configuration. Consequently, all possibilities would have to be provided and tested, which is too costly in most cases according to system engineers at BMW [Ebe09]. There are many ideas and, in parts, already detailed concepts for the interconnection of driver assistance systems, but most of them never make it into serial production.

6.1.6 Threats to Validity

The threat to internal validity is an experimenter bias, as the author performed the case study herself. External validity is threatened by specifics of the application domain as the case study was performed only in the embedded systems domain, although the method was successfully applied to a number of small examples from the information systems domain.

Countermeasure for both threats is further validation in a case study performed by industrial software developers. A follow-up case study in a different application domain performed by industrial research partners is also intended, as this would further show domain-independent applicability.

6.1.7 Summary

The case study showed the applicability of the DeSyRe approach for the driver assistance systems specification. The method for decomposition and refinement of system requirements was applied successfully in combination with an artifact model for requirements engineering. The results are subsystem requirements specifications for the ACC subsystem and the RFW subsystem that comply with the overall driver assistance systems requirements.

6.2 Case Study on Usefulness

The general usefulness of an integrated approach dealing with the decomposition of systems and their requirements was motivated in Chap. 1 with work distribution and systematic requirements engineering and by the encouraging feedback from the interview partners in the study on the state of practice (Sec. 2.2). The specific usefulness of the approach developed in the work at hand was evaluated by giving a tutorial for practitioners and, subsequently, asking them to fill out a questionnaire. Again, the presentation structure is inspired by Runeson and Höst [RH09].

Research Objective. (template from [WRH00])

> *Analyze* the DeSyRe approach
> *for the purpose of* validation
> *with respect to the* usefulness
> *from the point of view of the* software developer
> *in the context of* general software systems development.

Study Design. The author presented a tutorial on the approach followed by handing out a questionnaire that the audience was asked to fill out straight away. The tutorial was designed as a slide presentation with emphasis on the concepts of system decomposition criteria and requirements refinement and an overview of the guiding process. The questionnaire was designed according to the template by Davis [Dav89]. The following statements could be rated in 6 degrees from *I strongly agree* to *I strongly disagree*:

1. Using DeSyRe improves the structuredness of requirements engineering for complex systems.
2. Using DeSyRe improves the completeness of subsystem requirements specifications.
3. Using DeSyRe improves the traceability of requirements.
4. Using DeSyRe eases system integration as the approach ensures that all relevant information is captured and processed.
5. Using DeSyRe improves the reusability of requirements.
6. I would find DeSyRe useful for my work.

Execution. The tutorial was held at a small software development company and took about 75 minutes including discussion. The number of participants for the tutorial was 12. The audience took active interest in the tutorial and there was a lively discussion on details of the approach.

The number of filled out questionnaires was 11, as one participant decided he could not make any judgement due to lack of experience. He was a 16-year old apprentice who had just started the job. The other eleven participants were experienced software developers with academic background (diploma or master's degree in computer science, one in electrical engineering) and between 2 and 10 years of industrial project experience. Of the 11 handed in questionnaires, not all were completely filled out. The results are given in Tab. 6.7.

Discussion. Overall, the participants had the impression that the approach provided improvements with respect to all points, but they estimated the usefulness for their own daily work as rather low. The reason for the latter is presumably the generally low importance the developers gave to requirements

Table 6.7: Results of the Questionnaire on Perceived Usefulness

	strongly agree	agree	partially agree	partially disagree	disagree	strongly disagree	sum
Improved Structuredness	1	9	1				11
Improved Completeness	1	5	2	2			10
Improved Traceability	2	4	2	2			10
Eased Integration	3	1	4	2	1		11
Improved Reusability		3	4	2			9
Useful for My Work		2	3	2	3	1	11

engineering. According to their own perception, their projects are developed rather solution-oriented with weak attention to RE. This raises questions with regard to the suitability of the target group.

In contrast, structuredness, completeness, traceability, integration and reusability were all perceived as improving by means of the approach.

With regard to the overall research objective for the approach, the *improvement of structuredness* is the most emphasized characteristic as the intention was to find a *systematic* approach for the derivation of subsystem requirements specifications:

> *How can we systematically derive subsystem requirements specifications from system requirements specifications?* (Sec. 1.2)

Therefore, the rating of for the improvement of structuredness is an indicator for having achieved this objective.

Threats. A threat to internal validity may be the discussions that took place during the tutorial, as the participants may have influenced each other and would have answered differently after participating in the tutorial without sharing their thoughts in between.

The major threat to external validity is that the participants may be plain wrong, since they have not tried the method themselves. The counter measure taken during the tutorial was to use examples from real specifications so the audience could relate them to their experience as most of them have experience in software development for the application domain of the examples, i.e., embedded systems.

Another threat to external validity is that there is no systematic requirements engineering approach established within the company. Therefore, most of the developers have limited knowledge about requirements engineering in general and might not be the most appropriate audience to judge the approach after only a short tutorial. On the other hand, this might actually not harm transferability as requirements engineering in practice and, even more, its education and training, are often neglected in favor of other issues.

Conclusion. The results from the questionnaire on perceived usefulness indicate that the approach achieves the objective of improving the structuredness of results during the process of requirements engineering.

Summary. This chapter presented the evaluation and assessment of the DeSyRe approach. The evaluation of applicability was performed with a case study in the automotive domain on driver assistance systems. The evaluation of usefulness was performed with a questionnaire filled out by the participants of a tutorial on the method.

Chapter 7

Conclusion and Future Work

This chapter concludes the thesis at hand, gives a summary of the presented work, and provides an outlook on future work.

7.1 Summary of Results

This section summarizes the contributions of this work and denotes how the research questions from the introduction are covered by the DeSyRe method.

The motivation for this thesis lies in complex systems' development, where a development company specifies the overall system and assigns subcontractors for the distributed realization of the subsystems. For a smooth development without communication overhead in terms of numerous additional information requests by the subcontractor as well as successful integration of the subsystems following such an assignment, a subsystem specification is to be systematically derived from the system specification and includes all relevant information. This challenge was affirmed by an interview study on the state of practice (Sec. 2.2).

From the motivation, the central research question that was phrased is:

> *What is a good way for a requirements engineer to systematically derive* sub*system requirements specifications from system requirements specifications?*

To break it down, three more research questions were deduced from the central one: The first one about how to get an initial system decomposition, the second one about how to deduce subsystem requirements, and the third one about how to apply both steps during the development process (Sec. 1.2).

Contributions

So far, although the problems considered by the research questions are recognized (see also interviews in Sec. 2.2), yet missing is an approach that tackles these problems. In the thesis at hand, the contributions according to the research questions are the following:

Decomposition criteria catalogue. The guidance for decomposition is provided by a decomposition criteria catalogue that lists all potential influences on a system's decomposition (Chap. 3). The four main categories of criteria, namely *directive*, *functional*, *quality*, and *technical*, are further refined and detailed, including their source of information and stakeholders, and illustration with examples. The catalogue serves as extensive checklist for requirements engineers eliciting and gathering information and system architects reviewing the information before deciding on the design. This contribution answers research question 1: *What is a good way for the system architect to obtain the initial system decomposition?*

Subsystem Model and Requirements Refinement. The defined subsystem model serves as basis for a discussion of subsystem distribution across abstraction levels (Sec. 4.3). The subsystem requirements refinement is performed using patterns and assumption / guarantee specifications (Sec. 4.4). The patterns are a general pattern, and two special cases, *pipeline* and *subservice*, which are reduced versions of the general pattern that already cover a major part of requirements.

There are three alternatives for the treatment of nonfunctional requirements, equal to the handling of functional requirements in case of compositionality, inclusion of an adequate model for conditional compositionality (e.g., probabilities), or constraint handling in case the requirement is not decomposable. Furthermore, a tracing approach is described. This contribution answers research question 2: *What is a good way for the requirements engineer to deduce subsystem requirements from system requirements?*

DeSyRe Process. The process (Chap. 5) guides the decomposition of a system by use of the decomposition criteria catalogue, then the capturing of requirements and related information within an artifact model, the refinement of requirements and the extraction of a subsystem specification. This contribution answers research question 3: *How do the requirements engineer and the system architect perform both the decomposition and deduction during the requirements specification development process?*

Evaluation. The method is evaluated in a case study with respect to applicability and with an opinion survey with respect to usefulness after a tutorial on the approach in another case study. The results of the case studies (Sec. 6.1) showed that the method is applicable and the results of the questionnaire (Sec. 6.2) rated that the approach improves structuredness, completeness, traceability, integration and reusability of requirements.

7.2 Current and Future Work

There are some issues related to the work at hand that are either already in work or worth future investigation. These include further investigation on nonfunctional requirements, different application domains, repeated hierarchical decomposition, maintenance, and refactoring. Further issues are continued empirical validation and tool support.

Nonfunctional Requirements. The discussion of nonfunctional requirements in Sec. 4.7 provides a starting point for an extensive study on different quality attributes. One objective is an analysis of their potential transformation into functional requirements and the other one an analysis of particular quality attributes' compositionality.

Application Domains. The approach was evaluated with a case study from the embedded systems domain, but it is not designed to exclusively satisfy specifics of only this domain. Therefore, it is also interesting to analyze other domains like the information systems domain with respect to the applicability of DeSyRe. Neither the patterns nor the paradigm of assumption / guarantee are coupled to the embedded systems domain, therefore the expectation is that the approach proves to be equally applicable in other domains.

Hierarchical Decomposition. The approach is assumed to work hierarchically such that, in the first step, subsystems are deduced and specified and, in the second step, *subsub*systems are deduced and specified. Thereby, the requirements specifications for *subsub*systems are derived in the same manner as for the subsystems. However, a case study on that hypothesis might reveal whether repeated and hierarchical execution leads to additional implications for the development process.

Integration into Methodical Context. An important subsequent step to improve the usability and applicability of the DeSyRe approach, is to integrate it into a surrounding methodical environment. In this case, the methodical environment constitutes of further methods developed by the research group of Software & Systems Engineering at TUM. Therefore, current work is to relate the approach to the artifact model by Méndez [MFK09]. A first step towards integration with further requirements engineering activities is described in [BFI$^+$09].

Process Integration. The process described in Chap. 5 gives only general guidance on how to proceed, but does not detail on specific milestones and interplay with further activities of the overall software development life cycle. Therefore, an integration with the V-Modell XT [Bun08b] would provide practical plug-in possibilities for the DeSyRe approach with respect to its application within surrounding established activities. That way it could be listed with a number of other compatible methods in a ready-to-use method kit.

Maintenance and Evolution. As many projects continue for a long time after their first release, it is interesting which part of the requirements engineering artifacts is most important and relevant to keep up to date while at the same time avoiding overhead. Therefore, one task is to investigate the most relevant contents for maintenance and evolution, and another task is to find the right quality attributes for maintainable requirements engineering artifacts plus, subsequently, concrete guidance on how to maintain them over time.

Refactoring. A further interesting point to investigate is the difference between inlining and extracting subsystems. The question is, what the implications for the subsystem requirements specification are in case the subsystem shall be inlined and integrated into other systems. For systems developed using the same approach to requirements engineering, integration is smooth, but a case study on how to integrate subsystem specifications developed with different approaches might reveal interesting insights on the compatibility of specifications.

Empirical Validation. For the approach as presented in this work, as well as for the possible future extensions, further empirical evaluation is an important issue to validate applicability and usability in practice.

Tool Support. Finally, an analysis of how far the approach can be supported by tools reveals automatization potential for the decomposition of systems as well as the decomposition of requirements. Automated support for the decomposition criteria catalogue can be provided by offering the catalogue as checklist where the user can mark all relevant criteria and, as result, receives either a short summary (simple version of support) or even concrete design proposals (sophisticated version).

Tool support for the refinement and decomposition of requirements for a subsystem can be provided by an application that automatically provides templates for the assumption / guarantee specification of the system requirements and then their decomposition by selection of the pattern. The simple version would be that the user has to fill out the template, the more sophisticated version that the application proposes a decomposition and the user has to verify it.

Bibliography

[AL91] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253 – 284, 1991.

[AL94] Martín Abadi and Leslie Lamport. Decomposing specifications of concurrent systems. In *PROCOMET*, pages 327–340, 1994.

[AL95] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17:507–533, 1995.

[AS87] B. Alpern and F.B. Schneider. Recognizing safety and liveness. *Distributed computing*, 2(3):117–126, 1987.

[AUT06] AUTOSAR consortium. AUTOSAR - Automotive Open Systems Architecture. www.autosar.org, 2006.

[BB98] PerOlof Bengtsson and Jan Bosch. Scenario-based Software Architecture Reengineering. In *Proceedings of the International Conference on Software Reuse.*, pages 308–317, 1998.

[BBC$^+$00] Felix Bachmann, Len Bass, Gary Chastek, Patrick Donohoe, and Fabio Peruzzi. The architecture based design method. Technical Report CMU/SEI-2000-TR-001, CMU SEI Pittsburgh, 2000.

[BBC$^+$06] Len Bass, John Bergey, Paul Clements, Paulo Merson, Ipek Ozkaya, and Raghvinder Sangwan. A comparison of requirements specification methods from a software architecture perspective. Technical Report CMU/SEI-2006-TR-013, CMU SEI, 2006.

[BBH$^+$09] Peter Braun, Manfred Broy, Frank Houdek, Matthias Kirchmayr, Mark Müller, Birgit Penzenstadler, Klaus Pohl, and Thorsten Weyer. Entwicklung eines Leitfadens für das Requirements Engineering softwareintensiver Eingebetteter Systeme. Technical report, Technische Universität München, 2009.

[BCK03] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice.* Addison-Wesley, second edition edition, 2003.

[BDS08] U. Becker, J. Drewes, and E. Schnieder. Accident based requirements analysis for advanced driver assistance systems. *Analysis, Design, and Evaluation of Human-Machine Systems*, 10:1, 2008.

[Ben08] Gerd Beneken. *Logische Architekturen - Eine Theorie der Strukturen und ihre Anwendung in Dokumentation und Projektmanagement*. PhD thesis, Technische Universität München, 2008.

[BFI$^+$09] Manfred Broy, Andreas Fleischmann, Shareef Islam, Leonid Kof, Klaus Lochmann, Christian Leuxner, Birgit Penzenstadler, Daniel Méndez Fernández, Wassiou Sitou, and Sebastian Winter. Towards an Integrated Approach to Requirement Engineering. techreport, Technische Universität München, 2009.

[BGG$^+$06] Nadine Bramsiepe, Eva Geisberger, Johannes Grünbauer, Günter Halmans, Nadine Heumesser, Frank Houdek, Hannes Omasreiter, Wassiou Sitou, and Thorsten Weyer. REMsES D1.2: Stand von Praxis und Wissenschaft und Anforderungen an den Leitfaden. Project Deliverable, 2006.

[BGG$^+$07] Nadine Bramsiepe, Eva Geisberger, Johannes Grünbauer, Günter Halmans, Birgit Penzenstadler, Tim Schmidt, Ernst Sikora, Wassiou Sitou, and Thorsten Weyer. REMsES D2.2: Grobes Produktmodell inklusive der Abstraktionsebenen zur Strukturierung und Modellierung von Anforderungen. Project Deliverable, 2007.

[BGL$^+$08] Nadine Bramsiepe, Johannes Grünbauer, Klaus Lochmann, Birgit Penzenstadler, Tim Schmidt, Ernst Sikora, Wassiou Sitou, and Thorsten Weyer. REMsES D-3.2: Ausarbeitung des Leitfadens auf Abstraktionsstufe Funktionsgruppen. Project Deliverable, 2008.

[BH07] Manfred Broy and James Herbsleb, editors. *Global Software Development Handbook*. Auerbach Publications, 2007.

[BKM07] Manfred Broy, Ingolf Krüger, and Michael Meisinger. A formal model of services. *ACM Transactions on Software Engineering Methodology (TOSEM)*, 16(1), 2007.

[BP01] Manuel Brandozzi and Dewayne E. Perry. Transforming goal oriented requirements specifications into architecture prescriptions. In *Workshop "From Software Requirements to Architectures" STRAW*, 2001.

[Bro95] Manfred Broy. A functional rephrasing of the assumption/commitment specification style. Technical report, Technische Universität München, 1995.

[Bro05] Manfred Broy. Service-oriented Systems Engineering: Specification and Design of Services and Layered Architectures — The Janus-Approach. In Manfred Broy, Johannes Grünbauer, David Harel, and Tony Hoare, editors, *Engineering Theories of Software Intensive Systems*, volume 195. NATO Advanced Study Institute, 2005.

[Bro07] Manfred Broy. Model-driven architecture-centric engineering of (embedded) software intensive systems: modeling theories and architectural milestones. *Innovations in Systems and Software Engineering*, 3(1):75–102, 2007.

[Bro10] Manfred Broy. Towards a Theory of Architectural Contracts: Schemes and Patterns of Assumption/Promise Based System Specification. In *Software and Systems Safety: Specification and Verification*, 2010.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces and Refinement.* Springer New York, 2001.

[BS06] Jason Bloomberg and Ronald Schmelzer. *Service Orient or to be doomed.* Wiley & sons, 2006.

[BSW$^+$08] Manfred Broy, Bernhard Schätz, Doris Wild, Martin Feilkas, Judith Hartmann, Alexander Gruler, Birgit Penzenstadler, Johannes Grünbauer, and Alexander Harhurin. Umfassendes Architekturmodell für das Engineering eingebetteter software-intensiver Systeme. Technical Report TUM-I0816, Technische Universität München, June 2008.

[Bun07] Bundesregierung. Straßenverkehrsordnung. Bundesgesetzblatt Teil I, November 2007.

[Bun08a] Bundesamt für Sicherheit in der Informationstechnik. BSI-Standard 100-2: IT-Grundschutz-Vorgehensweise, Version 2.0. www.bsi.bund.de/gshb, 2008.

[Bun08b] Bundesamt für Sicherheit in der Informationstechnik. V-Modell XT. http://www.v-modell-xt.de/, 2008.

[Bun08c] Bundesregierung. Straßenverkehrs-Zulassungs-Ordnung. Bundesgesetzblatt Teil I, May 2008.

[Bun08d] Bundesrepublik Deutschland. EMV Gesetz. Bundesgesetzblatt, 2008.

[BV06] Susanne Briest and Mark Vollrath. In welchen Situationen machen Fahrer welche Fehler? Ableitung von Anforderungen an Fahrerassistenzsysteme durch In-Depth-Unfallanalysen. *VDI-Berichte*, 1960(1960):449–463, 10 2006.

[CA09] B.H.C. Cheng and J.M. Atlee. Current and Future Research Directions in Requirements Engineering. In *Design Requirements Engineering: A Ten-Year Perspective, Workshop, Cleveland, 2007*, page 11. Springer, 2009.

[Can01] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS '01: Proceedings of the 42nd IEEE symposium on Foundations of Computer Science*, page 136, Washington, DC, USA, 2001. IEEE Computer Society.

[Cas99] Jim Cash. All the key questions. *Roadfly. The complete automotive resource for buyers, sellers, and owners like you.*, Oct, 1999.

[Cle94] Paul Clements. From domain models to architectures. In *Workshop on Software Architecture, USC Center for Software Engineering*, 1994.

[CNYM00] Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-functional Requirements in Software Engineering.* Kluwer Academic Publishers, 2000.

[Coc00] Alistair Cockburn. *Writing Effective Use Cases.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[Con68] M. Conway. How do committees invent? *Datamation Journal*, pages 28–31, April 1968.

[Cre03] John Creswell. *Research Design: Qualitative, quantitative and mixed approaches.* SAGE Publications, 2003.

[DAC99] Matthew Dwyer, George Avrunin, and James Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the International Conference on Software Engineering*, 1999.

[Dav89] F.D. Davis. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS quarterly*, 13(3):319–340, 1989.

[Dav93] Alan M. Davis. *Software requirements: objects, functions, and states.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[dCP08] David Bettencourt da Cruz and Birgit Penzenstadler. Designing, documenting, and evaluating software architecture. Technical Report TUM-I0818, Technische Universität München, 2008.

[DDB$^+$05] Ovidiu Drugan, Ioanna Dionysiou, David Bakken, Thomas Plagemann, Carl Hauser, and Deborah Frincke. On the importance of composability of ad hoc mobile middleware and trust management. In Miroslaw Malek, Edgar Nett, and Neeraj Suri, editors, *Service Availability*, volume 3694 of *Lecture Notes in Computer Science*, pages 149–163. Springer, 2005.

[Deu09] Deutscher Verkehrssicherheitsrat. Bester Beifahrer - Fahrerassistenzsysteme, Innovationen für Sicherheit. online available at http://www.bester-beifahrer.de, 2009.

[DH] Jorge L. Díaz-Herrera. Embedded systems product lines: Process and models. website, project YES, http://cse.spsu.edu/yes/YES-PL.pdf.

[Dij68] Edsger W. Dijkstra. The structure of the THE-multiprogramming system. *Commun. ACM*, 11(5):341–346, 1968.

[DJH+08] Florian Deissenboeck, Elmar Juergens, Benjamin Hummel, Stefan Wagner, Benedikt Mas y Parareda, and Markus Pizka. Tool support for continuous quality control. *IEEE Software*, 25 (5):60–67, 2008.

[DKK+05] Joerg Doerr, Daniel Kerkow, Tom Koenig, Thomas Olsson, and Takeshi Suzuki. Non-functional requirements in industry - three case studies adopting the ASPIRE NFR method. Technical Report 025.05/E, Fraunhofer IESE, 2005.

[DL05] Norman Denzin and Yvonna Lincoln, editors. *The Sage Handbook of Qualitative Research.* Thousand Oaks, CA: Sage, 2005.

[DVM+05] W. Damm, A. Votintseva, A. Metzner, B. Josko, T. Peikenkamp, and E. Böde. Boosting re-use of embedded automotive applications through rich components. *Foundations of Interface Technologies. Elsevier Science*, August, 2005.

[Ebe09] Jörg Ebeling. Email about Interaction Examples for Driver Assistance Systems at BMW. unpublished, March 2009.

[EGHB09] A. Egyed, P. Grünbacher, M. Heindl, and S. Biffl. Value-based requirements traceability: Lessons learned. *Design Requirements Engineering: A Ten-Year Perspective*, pages 240–257, 2009.

[EGM01] Alexander Egyed, Paul Grünbacher, and Nenad Medvidovic. Refinement and Evolution Issues in Bridging Requirements and Architecture — The CBSP Approach. In *International SofTware Requirements to Architecture Workshop*, 2001.

[Eur06] Eurobarometer. Use of Intelligent Systems in Vehicles. Special Eurobarometer 267/Wave 65.4, 12 2006. available at http://www.bester-beifahrer.de/downloads.html.

[EYA+05] S. Easterbrook, E. Yu, J. Aranda, Yuntian Fan, J. Horkoff, M. Leica, and R.A. Qadir. Do viewpoints lead to better conceptual models? An exploratory case study. In *Proceedings of the International Conference on Requirements Engineering*, 2005.

[Fei10] Martin Feilkas. *Implizite Entwurfsregeln in Softwaresystemen: Entstehung, Erfassung und Konformitätsprüfung.* PhD thesis, Technische Universität München, 2010.

[FFH+09a] M. Feilkas, A. Fleischmann, F. Hölzl, C. Pfaller, K. Scheidemann, M. Spichkova, and D. Trachtenherz. A Top-Down Methodology for the Development of Automotive Software. Technical report, Technische Universität München, 2009.

[FFH+09b] Martin Feilkas, Andreas Fleischmann, Florian Hölzl, Christian Pfaller, Sabine Rittmann, Kathrin Scheidemann, Maria Spichkova, and David Trachtenherz. A Top-Down Methodology for the Development of Automotive Software. Technical Report I0902, Technische Universität München, 2009.

[FLB06] José Luiz Fiadeiro, Antónia Lopes, and Laura Bocchi. A Formal Approach to Service Component Architecture. *Web Services and Formal Methods*, 4184:193–213, 2006.

[Fle08] Andreas Fleischmann. *Modellbasierte Formalisierung von Anforderungen für eingebettete Systeme im Automotive-Bereich*. PhD thesis, Technische Universität München, 2008.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, 1995.

[GBB$^+$06] Eva Geisberger, Manfred Broy, Brian Berenbach, Juergen Kazmeier, Daniel Paulish, and Arnold Rudorfer. Requirements Engineering Reference Model (REM). Technical report, Technische Universität München, 2006.

[GF94] O. C. Z. Gotel and C. W. Finkelstein. An analysis of the requirements traceability problem. In *International Conference on Requirements Engineering*, pages 94–101, 1994.

[GGRS08] Frank Großhauser, Frank Gesele, Stephan Reichelt, and Karsten Schmidt. In die Realität überführt: Nutzung von AUTOSAR in der Serie bei Audi. *Automotive Elektronik*, 2008.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[Gli07] M. Glinz. On non-functional requirements. *Proceeddings of the International Conference on Requirementes Engineering*, 7:21–26, 2007.

[GRM04] Martin Große-Rhode and Stefan Mann. Model-based Systems Engineering in the Automobile Industry: Positions and Experiences. In *Proceedings of the Software Product Line Conference 2004, Workshop Automotive Software Architectures*, 2004.

[Grü08] Johannes Grünbauer. Feature Interactions auf Nutzungsebene. *Softwaretechnik-Trends*, 1, 2008.

[GY01a] Daniel Gross and Eric Yu. From non-functional requirements to design through patterns. *Requirements Engineering*, 6(1):18–36, 2001.

[GY01b] Daniel Gross and Eric S. K. Yu. Evolving system architecture to meet changing business goals: an agent and goal-oriented approach. In *International Software Requirements to Architecture Workshop*, 2001.

[Gyö08] Andreas Györy. Tracing zwischen Anforderungen und Design in mechatronischen Systemen. Master's thesis, Technische Universität München, 2008.

[HB07] M. Heindl and S. Biffl. An Initial Tracing Activity Model to Balance Tracing Agility and Formalism. Technical report, Vienna University of Technology, 2007.

[HB08] Matthias Heindl and Stefan Biffl. Modeling of requirements tracing. In Bertrand Meyer, Jerzy Nawrocki, and Bartosz Walter, editors, *Balancing Agility and Formalism in Software Engineering*, volume 5082 of *Lecture Notes in Computer Science*, pages 267–278. Springer, 2008.

[HDS$^+$07] Jane Hayes, Alex Dekhtyar, Senthil Sundaram, E. Holbrook, Sravanthi Vadlamudi, and Alain April. REquirements TRacing On target (RETRO): improving software maintenance through traceability recovery. *Innovations in Systems and Software Engineering*, 3:193–202, 2007.

[HG99] James D. Herbsleb and Rebecca E. Grinter. Splitting the organization and integrating the code: Conway's law revisited. In *Proceedings of the 21st international conference on Software engineering*, 1999.

[HMRR06] Jens Heidrich, Jürgen Münch, William E. Riddle, and Dieter Rombach. *New Trends in Software Process Modelling*, chapter People-Oriented Capture, Display, and Use of Process Information, pages 121–180. World Scientific, 2006.

[HNS00] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture*. Addison-Wesley, 2000.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576 – 580, 1969.

[HPP06] Andrea Herrmann, Barbara Paech, and Damian Plaza. ICRAD: An integrated process for the solution of requirements conflicts and architectural design. *International Journal of Software Engineering and Knowledge Engineering*, 16:1–34, 2006.

[HQR98] Thomas A. Henzinger, Shaz Qadeer, and Sriram Rajamani. *You assume, we guarantee: Methodology and case studies (Computer Aided Verification)*, volume 1427/1998, pages 440–451. Springer, 1998.

[HQRT02] Thomas A. Henzinger, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. An assume-guarantee rule for checking simulation. *ACM Trans. Program. Lang. Syst.*, 24(1):51–64, 2002.

[IEE98] IEEE. IEEE Recommended Practice for Software Requirements Specifications (IEEE Std 830-1998), 10 1998.

[Int01] International Standardization Organization. ISO9126 - International Standard for the Evaluation of Software Quality. http://www.iso.org, 2001.

[ITU96] ITU-TS Recommendation Z.120. Message Sequence Chart (MSC). International Telecommunication Union ITU-TS, Geneva, 1996.

[Jan97] Theo M.V. Janssen. *Handbook of logic and linguistics*, chapter Compositionality. MIT Press, 1997.

[JM90] F. Jay and R. Mayer. IEEE standard glossary of software engineering terminology. *IEEE Std*, 610:1990, 1990.

[JRvdL00] Mehdi Jazayeri, Alexander Ran, and Frank van der Linden. *Software Architecture for Product Families.* Addison-Wesley, 2000.

[KK99] R. Kazman and M. Klein. Attribute-based architectural styles. Technical Report 22, CMU SEI, 1999.

[KNR05] Marco Kuhrmann, Dirk Niebuhr, and Andreas Rausch. Application of the V-Modell XT - Report from A Pilot Project. In *Unifying the Software Process Spectrum, International Software Process Workshop*, pages 463–473. Springer, 2005.

[Kosed] Dagmar Koss. *Kompatibilität und Kompatibilitätsmanagement.* PhD thesis, Technische Universität München, to be published.

[Kru95] Philippe Kruchten. Architectural blueprints—The "4+1" view model of software architecture. *IEEE Software*, 12(6):42–50, November 1995.

[Kru00] P. Kruchten. *The rational unified process: an introduction.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2000.

[KS98] G. Kotonya and I. Sommerville. *Requirements Engineering.* John Wiley & Sons Ltd, 1998.

[Kuh08] Marco Kuhrmann. *Konstruktion modularer Vorgehensmodelle.* PhD thesis, Technische Universität München, jul 2008.

[Kul06] Kullmann. *Produkthaftungsgesetz.* Erich Schmidt Verlag, 5 edition, 2006.

[LCJZ08] Anders Mikael Lindgren, Fang Chen, Patrick W. Jordan, and Haixin Zhang. Requirements for the design of advanced driver assistance systems — the differences between swedish and chinese drivers. *International Journal of Design*, 2:41–54, 2008.

[LD06] Marco Lormans and Arie Van Deursen. Can lsi help reconstructing requirements traceability in design and test? In *Conference on Software Maintenance and Reengineering*, 2006.

[LFBW10] Markus Luckey, Daniel Méndez Fernández, Andrea Baumann, and Stefan Wagner. Reusing security requirements using an extended quality model. In *Workshop SESS at the IEEE International Conference on Software Engineering*, 2010.

[LL91] Kim G. Larsen and Lin Xinxin Liu. Compositionality through an operational semantics of contexts. *Journal of Logic and Computation*, 1(6):761–795, 1991.

[LO10] Jörg Leuser and Daniel Ott. Tackling semi-automatic trace recovery for large specifications. In Roel Wieringa and Anne Persson, editors, *Requirements Engineering: Foundation for Software Quality*, volume 6182 of *Lecture Notes in Computer Science*, pages 203–217. Springer Berlin / Heidelberg, 2010.

[Lon04] Lonn et al. FAR EAST: Modeling an automotive software architecture using the EAST ADL. *IEE Seminar Digests*, 2004.

[LS04] J. Leohold and C. Schmidt. Communication requirements of future driver assistance systems in automobiles. *Proceedings of the International Workshop on Factory Communication Systems*, pages 167–174, Sept. 2004.

[Luk00] K. Lukka. The key issues of applying the constructive approach to field research. *Management Expertise for the New Millenium. Publications of the Turku School of Economics and Business Administration, Series A-1*, 2000.

[LY01] Lin Liu and Eric Yu. From Requirements to Architectural Design — Using Goals and Scenarios. In *STRAW*, 2001.

[MBE01] Nenad Medvidovic, Barry W. Boehm, and Alexander Egyed. Refinement and Evolution Issues in Bridging Requirements and Architecture — The CBSP Approach. In *International SofTware Requirements and Architecture Workshop*, 2001.

[MFK09] D. Méndez Fernández and M. Kuhrmann. Artefact-based requirements engineering and its integration into a process framework. Technischer Bericht, Technische Universität München, 2009.

[MGEB03] Nenad Medvidovic, Paul Grünbacher, Alexander Egyed, and Barry W. Boehm. Bridging models across the software lifecycle. *Journal on Systems and Software*, 68(3):199–215, 2003.

[MHM87] P.W.G. Morris, G.H. Hough, and WG Morris. *The anatomy of major projects: A study of the reality of project management.* Wiley Chichester, UK, 1987.

[Mic06] Microchip Technology Inc. BMW Signs SmartShunt Technology Patent License Agreement with Microchip Technology. *Embedded Computing Design*, Jan, 2006.

[Mon70] Richard Montague. Universal grammar. *Theoria*, Vol. 36 Iss. 3:p. 373–398, 1970.

[MT00] N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, Jan 2000.

[Nav07] Elena Navarro. *ATRIUM Architecture Traced from Requirements by Applying a Unified Methodology.* PhD thesis, University of Castilla-La Mancha, 2007.

[Neuss] Philipp Neubeck. *Probability Extension of the Specification Method Focus.* PhD thesis, Technische Universität München, work in progress.

[NRP03] E. Navarro, I. Ramos, and J. Perez. Software requirements for architectured systems. In *Proceedings of the 11th IEEE International Requirements Engineering Conference*, 2003.

[Nus01] Bashar Nuseibeh. Weaving the software development process between requirements and architecture. In *International SofTware Requirements to Architectures Workshop*, 2001.

[Obj07] Object Management Group. UML 2. http://www.uml.org/#UML2.0, 2007.

[OMG04] OMG. Reusable asset specification. http://www.omg.org/technology/documents/formal/ras.htm, 2004.

[Par72] David L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[Par76] David Lorge Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2(1):1–9, 1976.

[Par79] David L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, 5(2):128–138, 1979.

[Par94] David L. Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering*, pages p.279–287, 1994.

[PBKS07] Alexander Pretschner, Manfred Broy, Ingolf H. Kruger, and Thomas Stauner. Software engineering for automotive systems: A roadmap. In *International Conference on Future of Software Engineering*, pages 55–71, 2007.

[PBP09] Birgit Penzenstadler, Peter Braun, and Jan Philipps. Deliverable 7.3: Anwendung des Leitfadens auf Werkzeuge im REM - Werkzeugkonzept. Project Deliverable, January 2009.

[PK08] Birgit Penzenstadler and Dagmar Koss. High confidence subsystem modelling for reuse. In *High Confidence Software Reuse in Large Systems, Proceedings of the Intl. Conf. on Software Reuse*, 2008.

[PL08] Birgit Penzenstadler and Joerg Leuser. Complying with Law for RE in the Automotive Domain. In *Workshop Requirements Engineering and Law (RELAW) at the Intl. Requirements Engineering Conference*, 2008.

[PM05] Jaime Pavlich-Mariscal. A framework for composable security definition, assurance, and enforcement. In *Doctoral Symposium of the Models Conference*, 2005.

[Poh07] Klaus Pohl. *Requirements Engineering.* dpunkt.verlag, 2007.

[PPV00] Dewayne E. Perry, Adam A. Porter, and Lawrence G. Votta. Empirical studies of software engineering: a roadmap. In *Proceedings of the International Conference on Software Engineering: The Future of Software Engineering*, pages 345–355, 2000.

[PS07a] Klaus Pohl and Ernst Sikora. COSMOD-RE: Supporting the co-design of requirements and architectural artifacts. In *International Conference on Requirements Engineering*, pages 258–261, 2007.

[PS07b] Klaus Pohl and Ernst Sikora. Structuring the co-design of requirements and architecture. In *International Working Conference on Requirements Engineering: Foundations for Software Quality*, pages 48–62. Springer, 2007.

[PSP09] Birgit Penzenstadler, Ernst Sikora, and Klaus Pohl. A requirements reference model for model-based requirements engineering in the automotive domain. In *International Working Conference on Requirements Engineering: Foundation for Software Quality*, 2009.

[PvKD$^+$03] Barbara Paech, Antje von Knethen, Joerg Doerr, J. Bayer, Daniel Kerkow, Ronny Kolb, A. Trendowicz, T. Punter, and A. Dutoit. An Experience-Based Approach for Integrating Architecture and Requirements Engineering. In *International SofTware Requirements to Architectures Workshop*, 2003.

[Ram98] B. Ramesh. Factors influencing requirements traceability practice. *Communications of the ACM*, 41(12):37–44, 1998.

[RDSS09] Robert Bosch GmbH, DaimlerChrysler AG, SSE Universität Duisburg-Essen, and Software and Systems Engineering Technische Universität München and Validas AG. Project REMsES. `http://www.remses.org`, 2009.

[Rei09] Günther Reichart. Bordnetze im Automobil: Aspekte der Hardware und Software, der Vernetzungstechnologien und der Systemarchitektur. Talk at “Methoden des Software Engineering”, Technische Universität München, April 2009.

[Res08] Markus Reschka. Discussions about Driver Assistance Systems at BMW. Personal interview notes, August 2008.

[RH09] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

[Ris07] Stefan Rischard. REMsES Illustrator "Radio-Frequenz-Warner (RFW)". Project Deliverable, 9 2007. AP6.

[Rit08a] Sabine Rittmann. Funktionale Anforderungen an das ACC mit Einparkassistenten. Project Deliverable, October 2008.

[Rit08b] Sabine Rittmann. *A methodology for modeling usage behavior of multi-functional systems.* PhD thesis, Technische Universität München, 2008.

[RJ01] B. Ramesh and M. Jarke. Toward reference models for requirements traceability. *IEEE Transactions on Software Engineering*, 27(1):58–93, 2001.

[Rob91] Robert Bosch GmbH. CAN Specification Version 2.0, September 1991. superseded by the standard ISO 11898.

[RR06a] Matthias Recknagel and Chris Rupp. Meßbare Qualität in Anforderungsdokumenten. *Automotive Vertikal*, 2:12–17, 2006.

[RR06b] James Robertson and Suzanne Robertson. Volere: Requirements specification template, 2006. `http://www.volere.co.uk/`.

[RR07] James Robertson and Suzanne Robertson. *Mastering the requirements process.* Addison-Wesley, 2007.

[RZ91] S.J. Russell and S. Zilberstein. Composing real-time systems. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 212–217, 1991.

[SACO02] Rachel Smith, George Avrunin, Lori Clarke, and Leon Osterwell. PROPEL: An approach supporting property elucidation. In *Proceedings of the International Conference on Software Engineering*, 2002.

[SB07] Klaus Steinhauser and Gunther Bauer. Hitech im Antriebsstrang: Vernetzte Funktionen und Entwicklung. *Hanser Automotive*, 10:28–32, 2007.

[SFGP05] Bernhard Schätz, Andreas Fleischmann, Eva Geisberger, and Markus Pister. Model-Based Requirements Engineering with AutoRAID. In *Workshop Modellbasierte Qualitätssicherung*, pages 511–516, 2005.

[Sha03] Mary Shaw. Writing good software engineering research papers. In *Proceedings of the 25th International Conference on Software Engineering*, pages 726–736. IEEE Computer Society, 2003.

[Sie02] Johannes Siedersleben. Quasar: Die sd&m Standardarchitektur. Technical report, sd&m AG, 2002.

[Sne98] Gregor Snelting. Paul Feyerabend und die Softwaretechnologie. *Informatik-Spektrum*, 21:273–276, 1998.

[Sof09] Software Engineering Institute Carnegie Mellon University. Capability maturity model integration. http://www.sei.cmu.edu/cmmi, 2009.

[SPI05] SPICE User Group. Software process improvement and capability determination. http://www.automotivespice.com/, 2005.

[Sta09] Statistisches Bundesamt Deutschland. Fahrzeugbestand Kraftfahrzeuge. published online at http://www.destatis.de, Jan 2009.

[Stü02] Rupert Stützle. *Wiederverwendung ohne Mythos: Empirisch fundierte Leitlinien für die Entwicklung wiederverwendbarer Software*. PhD thesis, Technische Universität München, 2002.

[SV03] Alberto L. Sangiovanni-Vincentelli. Electronic-system design in the automobile industry. *IEEE Micro*, 23(3):8–18, 2003.

[Sza07] Zoltán Gendler Szabó. Compositionality. Stanford Encyclopedia of Philosophy, First published Thu Apr 8, 2004; substantive revision Wed Feb 14, 2007. viewed on January 21st, 2009.

[TA05] J. Tyree and A. Akerman. Architecture decisions: demystifying architecture. *IEEE Software*, 22(2):19–27, March-April 2005.

[Uni03] United Nations Economic Comission for Europe. ECE R43, Revision 2. http://www.unece.org/, July 2003.

[VB06] Volkswagen AG and BMBF. Forschungsinitiative INVENT (Intelligenter Verkehr und Nutzergerechte Technik): Schlussbericht FAS, February 2006.

[VCK96] John Vlissides, James O. Coplien, and Norman L. Kerth. *Pattern Languages of Program Design 2*. Addison-Wesley, 1996.

[vdB00] Michael von der Beeck. Behaviour specifications: Equivalence and refinement. In H. Giese and S. Phillippi, editors, *Visuelle Verhaltensmodellierung verteilter und nebenläufiger Software-Systeme*, 2000.

[vKP02] Antje von Knethen and Barbara Paech. A survey on tracing approaches in practice and research. Technical report, IESE-Report, 095.01/E, 2002.

[vL00] Axel van Lamsweerde. Formal specification: a roadmap. In *Proceedings of the International Conference on Software Engineering: The Future of Software Engineering*, pages 147–159, 2000.

[vL01] Axel van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, page 249, 2001.

[vL03] Axel van Lamsweerde. From system goals to software architecture. In *School on Formal Methods*, pages 25–43, 2003.

[vL08] Axel van Lamsweerde. Requirements engineering: From craft to discipline. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 238–249, 2008.

[vL09] Axel van Lamsweerde. *Requirements Engineering: From system goals to UML models to software specifications.* Wiley & sons, 2009.

[WBB$^+$06] Rob Wojcik, Felix Bachmann, Len Bass, Paul Clements, Paulo Merson, Robert Nord, and Bill Wood. Attribute-driven design (ADD). Technical Report CMU/SEI-2006-TR-023, Carnegie Mellon University, Pittsburgh, 2006.

[WD07] Stefan Wagner and Florian Deissenboeck. An Integrated Approach to Quality Modelling. In *5th Workshop on Software Quality (5-WoSQ)*. IEEE Computer Society Press, 2007.

[Wiled] Doris Wild. *Modellbasierter Übergang von der logischen Architektur eingebetteter Systeme zur Plattform.* PhD thesis, Technische Universität München, to be published.

[WKV08] J. Werneke, A. Kassner, and M. Vollrath. An analysis of the requirements of driver assistance systems — when and why does the driver like to have assistance and how can this assistance be designed? *Proceedings of European Conference on Human Centred Design for Intelligent Transport Systems*, page pp. 193 Ũ 204, 2008.

[WP08] Thorsten Weyer and Klaus Pohl. Eine Referenzstrukturierung zur modellbasierten Kontextanalyse im Requirements Engineering softwareintensiver eingebetteter Systeme. In Thomas Kühne, Wolfgang Reisig, and Friedrich Steimann, editors, *Modellierung*, pages 181–197, 2008.

[WRH00] C. Wohlin, P. Runeson, and M. Höst. *Experimentation in software engineering: an introduction.* Springer Netherlands, 2000.

[ZJ97] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering Methodology*, 6(1):1–30, 1997.

Appendix A

ACC Appendix

A.1 *ACC Requirements*

There are 28 original informal requirements in natural language listed in [FFH$^+$09b].

1. *Select target vehicle. Probability is calculated by radar data. Condition: Within the defined range, High probability, Most close range vehicle.*
2. *Select target vehicle. Reject Parking cars.*
3. *Follow-up control of ACC system starts in case of target vehicle exists. Distance between two cars is controlled with in the target. Target acceleration is decided on deviation of distance and relative speed. Target acceleration is conveyed from Drive assist ECU to fusion ECU. Fusion ECU provides request to engine and brake components.*
4. *Follow-up control of ACC system starts in case of target vehicle exists. Distance between two cars is configurable depend on vehicle speed.*
5. *Constant speed control of ACC system starts in case of target vehicle NOT exists. Vehicle speed is controlled for target speed.*
6. *Target speed is configurable by Driver operation. Target speed is defined with cruise configuration switch. [Target speed] Initial condition: current speed, Increment: +5km/h, Decrement : -5km/h*
7. *Following distance is configurable by Driver operation. Following distance is changed by Setup SW. (3 Stage setup: Long $\rightarrow$ Middle $\rightarrow$ Short $\rightarrow$ Long $\rightarrow \ldots$)*
8. *ACC vehicle speed is in the defined range. ACC vehicle setting speed range: 50-100km/h*
9. *ACC acceleration is in the defined range. [ACC target acceleration range] Lower limit: -2.5m/s^2, Upper limit: 1.5m/s^2*
10. *Put ACC slowdown limit warning in action, when acceleration speed of ACC control is under lower limit. In case of ACC target acceleration is under limit, ACC slowdown limit warning is commanded.*
11. *ACC will be able to start, if vehicle speed is within the specified speed frame. ACC control will be able to start, if vehicle speed is between 45km/h and 110km/h.*

12. *When vehicle speed is under defined range, ACC control is terminated. ACC control will be terminated, if vehicle speed is less than 40km/h.*
13. *ACC control will start by drive SW operation. ACC control will start in case of pushing [SET SW] by driver.*
14. *ACC control will be terminated by drive SW operation. ACC control will be terminated in case of pushing [CANCEL SW] by driver.*
15. *Select Pre-Crash Safety (PCS) target. Set PCS target when scanned object has a high probability of collision. (Embedded on radar function)*
16. *Put PCS warning in action, when collision time is short. In case of collision time is shorter then defined range, PCS warning is in action.*
17. *When collision time is further short, PCS brake assist is set for ready condition. In case of collision time is shorter than defined range, PCS brake assist is set for ready condition.*
18. *When collision time is further short, PCS seat belt control is set for ready condition. In case of collision time is shorter then defined range, PCS seat belt control is set for ready condition.*
19. *When collision time is further short, PCS brake control is executed. In case of collision time is shorter than defined range, PCS brake control is executed.*
20. *When vehicle speed is in defined range, PCS comes in. When vehicle speed is over 30km/h, PCS comes in.*
21. *Drive's brake operation has a priority to ACC control. When Drive operate brake pedal, ACC control is terminated.*
22. *Driver's acceleration operation has a priority to ACC control. When Driver operate acceleration pedal, vehicle speed is able to speed up. (Embedded on engine ECU)*
23. *Driver's acceleration operation has a priority to ACC control. When Driver operate acceleration pedal, ACC brake control is suspended.*
24. *Compare request level of speed reduction between Driver's brake operation and PCS brake control. PCS brake control is continued when Driver operates brake pedal.*
25. *PCS brake control has a priority to Driver's accelerator pedal operation. During PCS brake control is in execution, Driver's acceleration request is turned down. (Embedded on engine ECU)*
26. *Compare request level of speed reduction between ACC control and PCS control. Strong request has a priority. Brake control has to be active if PCS brake control is NOT active.*
27. *Compare request level of speed reduction between ACC control and PCS control. Strong request has a priority. Engine throttle has to be closed if PCS Control is active.*
28. *ACC Control should be canceled when PCS brake control is operated.*

A.2 *Requirements Dependencies.*

For the definition of the dependency types see Sec. 5.3.

- Refinements: (2) refines (1), (4 and 7) refine (3), (6) refines (5), (14) refines (13), (16) refines (15), (17 - 19) refine (16), (23) refines (22), (27) refines (26).
- Concept bundles: (21 - 28) are the concept bundle “priority management”.
- Feature bundles: (1 - 7) are the feature bundle “follow-up”, (15 - 19) are the feature bundle “PCS warning”, (21 - 23) are the feature bundle “driver’s command”.
- Logical components: (1 - 14) are the logical component “ACC”, (15 - 20) are the logical component “PCS”
- Architectural bundles: The whole ACC including the PCS is one architectural bundle.
- There were no examples found for architectural restrictions, architectural alternatives, conflict negotiation, or architectural decisions.