A Service-Oriented Development Process for Distributed Reactive Systems

Michael Johannes Meisinger

Institut für Informatik
der Technischen Universität München

A Service-Oriented Development Process for Distributed Reactive Systems

Michael Johannes Meisinger

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

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Johann Schlichter
Prüfer der Dissertation:	
1.	Univ.-Prof. Dr. Dr. h.c. Manfred Broy
2.	Prof. Dr. Ingolf Krüger University of California, San Diego U.S.A.

Die Dissertation wurde am 15. September 2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 11. Januar 2016 angenommen.

Abstract

The complexity of modern software-intensive distributed reactive systems increases year by year; in large parts caused by a growing number of system functions spread across more system components, leading to more complex interactions. System deployments often occur in more distributed environments with increasingly heterogeneous technologies. The difficulty of developing and maintaining such systems is directly related to their complexity. The automotive domain provides telling examples for how complexity has become a major obstacle to further innovation and growth in system development. New techniques and optimized processes are necessary to maintain and increase the level of system quality while keeping development costs at bound.

Model-based development approaches have proven to keep the complexity of larger distributed systems manageable by providing expressive models with consistent, abstract views on relevant parts of the system. Successful model-based approaches have been applied to component-based and object-oriented systems; they are, however, challenged by multifunctional systems. Service-oriented methods promise to address these challenges, by formalizing the usage view of a system and its components in form of services. This emphasizes the functional dependencies and interaction behavior of system components and renders their inner working a well understood implementation concern, increasing confidence in the correctness of component interactions, and separating functional system behavior from technical design choices. In particular for larger projects, a systematic development process can complement and magnify the positive effects of such development methods. The process drives the creation of work products, manages their dependencies, and orchestrates the activities performed by the development team, ultimately ensuring timeliness and quality of delivery.

In this thesis, we describe promising techniques for managing system development complexity in a number of application domains. We distill a list of common key characteristics of successful development approaches and conclude that no existing approach satisfies all characteristics equally well. As a step towards a more comprehensive service-engineering solution that carefully advances current best practices, we introduce a service-oriented, model-based development process for distributed reactive systems, with primary focus on

the design of system architectures. The main contribution of our work is the process that brings together an existing, precisely defined service notion and service-oriented development method with our domain independent process artifact and activity model. We apply the definition of a service as the formalization of observable interaction behavior of a system or a component, in order to provide a specific function. We make use of the formal distributed system model FOCUS for the precise interpretation of service behavior. Our process treats services as first class elements of the system architecture, traceable through all phases of system development, beginning with requirements engineering. The process supports seamless definition and refinement of hierarchical system architectures using services, offering effective notations and systematic, iterative development steps applicable to many distributed reactive system architectures.

The center part of this thesis—its main contribution—is dedicated to introducing the process artifact and activity models. The comprehensive artifact model defines meaning and dependencies of all relevant development artifacts. It was designed to represent all concepts of the formal model. The activity model shows how these artifacts can be created, iteratively refined, and kept consistent in larger, distributed projects with many participants. To offer immediate practical applicability, we embed our process as a service-oriented extension of the system development framework V-Modell XT. Throughout this thesis, we make use of a running example.

As part of the evaluation, we describe a tool architecture supporting and automating our process. We compare our approach to related work in the literature, and assess it using our list of key characteristics. The evaluation shows that applying our approach leads to a more comprehensive understanding of the system under development, with a strong focus on its interactions, thereby positively influencing development productivity, outcome quality, and system maintainability, while limiting the variability of development duration and cost.

Kurzfassung

Wir beschreiben einen dienstbasierten Entwicklungsprozess für verteilte, reaktive Systeme mit Schwerpunkt auf der nahtlosen Designentwicklung von Systemarchitekturen und modellbasierten Designspezifikationen. Wir definieren ein Dienstkonzept und stellen dessen Vorteile bei der Entwicklung von verteilten, reaktiven Systemen dar. Dies während aller Entwicklungsphasen und insbesondere bei der Anwendung auf multifunktionale softwareintensive Systeme. Unser Entwicklungsprozess basiert auf einer reichhaltigen formalen Theorie zur Spezifikation verteilter Systeme und deren Architekturen. Die Evaluierung unseres Ansatzes zeigt dessen Vorteile in der Reduzierung der Entwicklungskomplexität, was zur einer systematischen Entwicklung mit reproduzierbaren Ergebnissen, hochwertigen Resultaten und beschränkten Kosten beiträgt.

Acknowledgments

This thesis would not have been possible without the extensive support of a number of people. First I would like to thank Prof. Dr. Dr. h.c. Manfred Broy for the opportunity to work in his research group. I am grateful for his advice and patience during the long time of completing this thesis, and for the opportunity to pursue significant parts of my research in San Diego. I would like to thank the DFG for sponsoring the research project "InServe" that supported this stay. I would also like to thank Prof. Dr. Ingolf Krüger at the University of California, San Diego, for inviting me into his research group and for the stimulating discussions we had over the years about services and software architecture. I am also grateful for his support enabling me to join UCSD as a software architect, and for agreeing to co-advise this thesis.

My deepest gratitude goes to my family. To my wife Mary, who supported me with her love and continuously encouraged me to complete this dissertation, freeing time when needed. And to our little son, Alexander Michael, who still allowed me to put the finishing touches on this document during his first year of age. I would like to dedicate this thesis to you, Alexander, and wish you a successful career. And to my parents, Renate and Johannes Meisinger, who sent me off to a good start.

I would like to thank my former colleagues in my research groups. In particular, I would like to thank David Cruz, Martin Deubler, Michael Gnatz, Johannes Grünbauer, Alexander Gruler, Markus Pister, Andreas Rausch, and Sabine Rittmann, then at the Technische Universität München, and Barry Demchak, Frederic Doucet, To-ju Huang, and Massimiliano Menarini at the University of California, San Diego. Thank you for stimulating discussions, good advice and friendship.

I would also like to thank my friends Joseph Brooks and Douglas Walker, who made me feel at home when I first arrived in San Diego, and who never never stopped reminding me to just "get this thesis done". Many thanks also to the project partners, colleagues and friends that supported me in this effort.

Contents

List of Figures

List of Tables

1 Introduction

This thesis introduces an iterative process for the service-oriented development of distributed reactive systems. The process is based on a formal system design approach. In this chapter, we motivate the need for such a process by highlighting the challenges system architects face when designing distributed reactive systems. We describe representative approaches from several application domains and analyze their strengths and weaknesses. We conclude that interaction complexity is one of the main impediments to high quality system development and that service-oriented model-based development promises to improve this situation, in particular when supported by an effective development process. We present our contribution—a service-oriented development process making use of a strong, formally founded service notion—to addressing these complexities. At the end of this chapter, we provide an outline for the remainder of this thesis and discuss related work.

Contents

1.1 Motivation

Software-intensive, distributed reactive systems are omnipresent in today's world albeit often invisible to their users. They occur in many forms: for instance as networks of electronic control units (ECUs) built into modern automobiles, as "cloud" based Internet platforms providing Web-scale applications to business and end users, and as "Big Data" processing cyberinfrastructures connected to large-scale sensor networks.

A *distributed system* is defined as "a collection of independent computers that appears to its users as a single coherent system" [TVS01]. A *distributed reactive system* is a distributed system that continuously interacts with its environment. A *software-intensive system* is any system where software contributes essential influences to the design, construction, deployment, and evolution of the system as a whole [Arc00]. A *multifunctional system* provides more than one function to its users [Bro10]. In this thesis, we focus on distributed reactive systems that are software-intensive and multifunctional.

Designing such systems is a difficult engineering task because of the high complexity caused by interacting system components and the large number of system states and conditions to consider. In the following sections, we describe some of the main engineering challenges. We investigate distributed systems from different application domains. Our running example is a telling representative of one of these systems; we introduce it below.

Designing viable system architectures is crucial to keep system interactions manageable and maintainable. We describe existing techniques, such as component-based, model-based and service-oriented development that have proven effective in developing viable architectures. We describe how systematic development processes can structure the development work into manageable tasks that each result in defined work products with verifiable properties.

These analyses will enable us to state the problem with distributed system development and propose a solution.

1.1.1 Challenges in Distributed System Development

Science Cyberinfrastructures This class of systems, commonly referred to as cyberinfrastructures [Dan03] (CI) or electronic science platforms, provide applications and infrastructure services to support science users, in particular to address "Big Data" [MCB+11] related challenges. Capabilities include data acquisition from sensor networks, real-time data distribution, data analysis and visualization, and high-performance computing (HPC) [WB09, Dan03, HT05, Fos05, CAF+09]. These platforms often manage a high number of heterogeneous physical resources—including sensors, telemetry systems, compute clusters, storage nodes—and electronic resources—such as real-time data feeds, data archives, visualization results, transformation algorithms, analysis tools, and virtual collaboration work spaces. Users and operators from different domains of authority and user communities provide,

share and access these resources, expecting the system to provide governance in form of consistent policy, security and error management. Science Cyberinfrastructures are often the result of system integration [HW03] at large to ultra-large scale [LJS08].

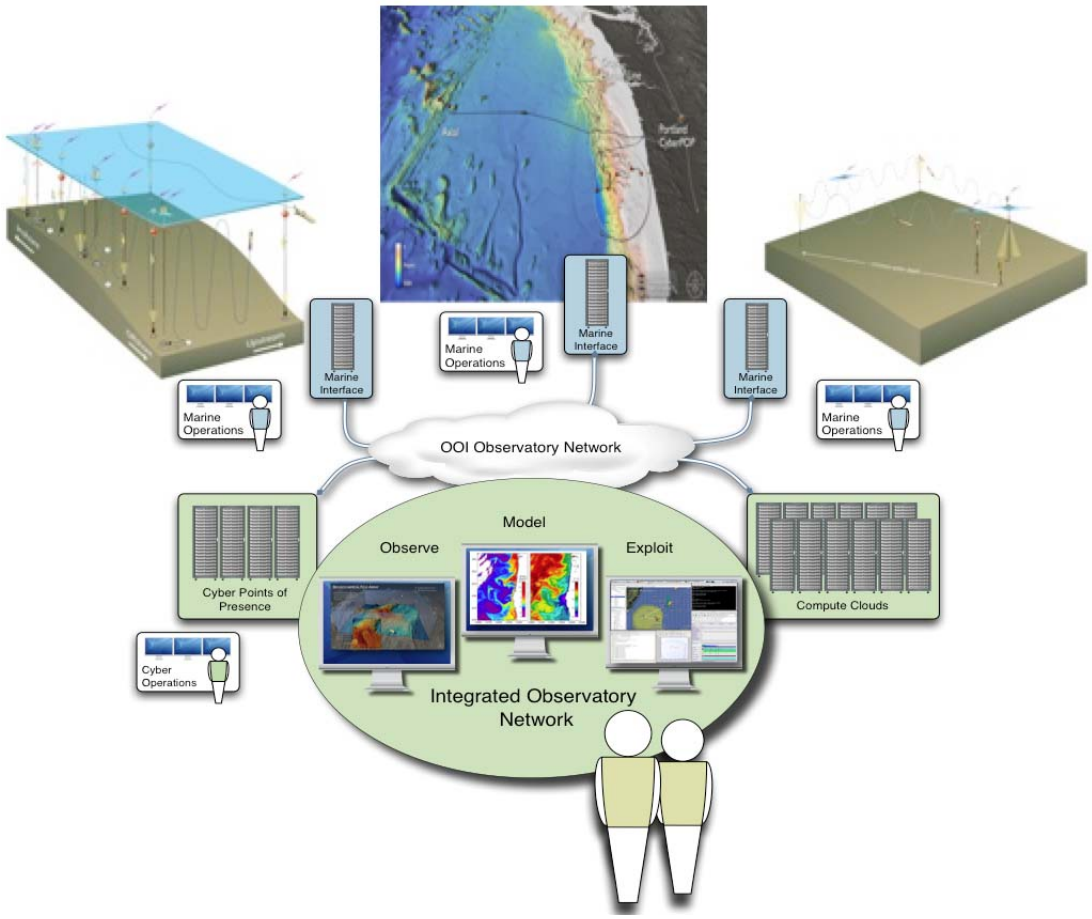

Figure 1.1: Ocean Observatories Initiative Cyberinfrastructure

Figure 1.1 illustrates the integrative nature of the U.S. Ocean Observatories Initiative (OOI) Cyberinfrastructure [Ini08, SGO+10, CAF+09]. The resulting system of systems, the "OOI Integrated Observatory Network" comprises observatories of different scale, as well as compute, network and storage resources with a sophisticated layer of software-enabled services. Science users can pursue the scientific activities of observing, analyzing data, and controlling the instrumentation. Observing includes accessing various transformed and aggregated science data products. Data analysis may mean running numerical simulation models using observation data from various sources and producing various forms of complex visualizations. Controlling the instrumentation may occur in real-time or based on a predefined mission plan, potentially triggered by an environmental event such as a hurricane. The system needs to coordinate shared resources in real-time and simultaneously protect long-running observations.

Figure 1.2 illustrates the various software capabilities provided OOI; it distinguishes multiple groups of users controlling different parts of the system. The figure shows "facilities" as independent domains of authority with instrumentation of multiple ownership. Data sets, storage and compute resources are associated with these facilities and a common set of services is available to all users. Different roles of users interact with the system. Policies apply consistently within each facility. Resources can be shared across facilities based on electronically maintained "contracts". A wide variety of heterogeneous science and infrastructure tools must be integrated into a consistent system.

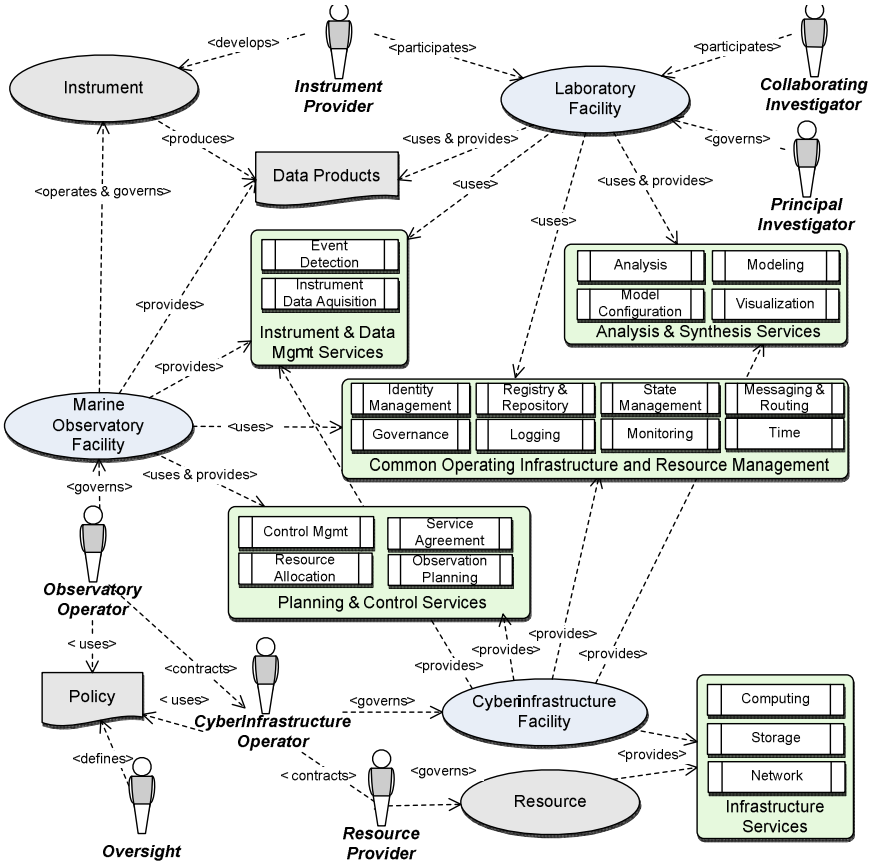

Figure 1.2: OOI Cyberinfrastructure capabilities and governance, from [Ini08]

The OOI Integrated Observatory Network is deployed at an extensive physical scale. Several physical sites provide compute, storage and networking resources. Additional compute and storage resources are available through an integration with academic and commercial cloud execution providers. A high-speed network connects these sites. Redundant hardware protects the system from hardware and network failure. The software components are distributed across the deployment sites, integrated using a messaging solution. The system is highly multifunctional and reactive in nature.

Designing a large-scale system such as the OOI Integrated Observatory Network is very complex. The system design must pay attention to the numerous interdependencies on the network, compute node, service infrastructure, data management and science application levels. Maintaining such a complex system design gets even harder, if requirements change over time or if project size and time span exceed the capacity of individual architects and developers, requiring larger teams with frequent communication and knowledge transfer. Successfully developing systems of this scale requires advanced software and system engineering methods.

Embedded Control Systems This class of systems exists, for instance, in modern automobiles, air planes and industrial production machinery. Systems are highly reactive, often distributed and heterogeneous, and come with high safety and real-time requirements [BKM08, Gri03, NP03, Lev00, BKT+06, Gro07]. Software and hardware components can be very intertwined and execution environments are often constrained by limited available resources, such as power, processor cycles, storage and communication bandwidth.

Modern automotive control systems are comprised of multiple embedded control units (ECU). Such units are linked via network buses and communicate by exchanging signals. The complexity of the overall system behavior can increase exponentially with the number of components involved or functions provided, and thereby signals exchanged. Some automobiles consist of up to 70 control units connected via up to three different bus systems. Vehicle safety and customer convenience functions require significant exchange of information between the control units. The central locking system (CLS) of a modern premium car is a good example. Figure 1.3 shows a CLS with its sensors, actuators and interacting control systems. It illustrates the complexities in embedded automotive control systems, caused by distributed interdependent functionalities. The depicted system components need to be orchestrated reliably to provide the Central Locking System (CLS) functionality to the end user, the driver of a modern car.

Besides the locking and unlocking behavior—already involving mechanical lock and remote control—functions need to exist to trigger lock/unlock depending on certain vehicle speeds and in case of impact. Locking and unlocking must be signaled to the user via the lighting system. Users expect personalized setup of the car seat and the multimedia system. Functionality for lock management, light control, crash sensing and the multimedia system is typically spread across multiple embedded control units; extensive interaction is required to provide almost all functions.

In the next section, we introduce a detailed example of an embedded control system as our running example.

Telecommunications Systems Systems out of this class provide the intelligent communication networks that are the backbone of our information society [Fre04, Zav93]. Early telecommunications systems mainly provided call-oriented telephony augmented by user

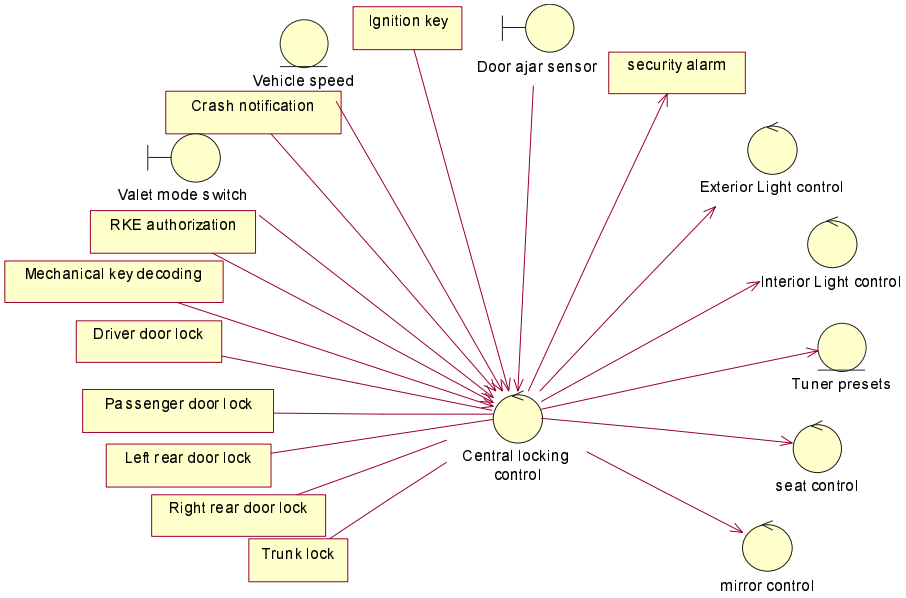

Figure 1.3: Central Locking System sensors, actuators, controllers and services, from [KNP04]

requested features; more recent systems realize the information highways of the present and future, such as for the Internet and the cloud networks.

Telecommunications systems as well as cloud and network infrastructure increasingly support mobile devices and mobile applications at unprecedented scale. These networks are massively distributed and need to be of high performance and reliability. Some telecommunications systems are highly multifunctional, providing functions ranging from network load balancing to content filtering, dynamic routing, content multi-casting, supporting multiple quality-of-service (QoS) levels [Day08].

Telecommunications systems traditionally have very high availability requirements. Systems must continue to function in the presence of hardware and network failures and support maintenance while operational. In some cases such as sensor networks and disaster response systems, with unreliable, intermittent wireless networks, the systems need to self-organize and dynamically manage scarce resources such as available bandwidth and power.

Business Information Systems These systems are mostly reactive in nature and provide a number of functions to execute business processes within an organization. This often requires complex data manipulations, dealing with many concurrent users and integrating heterogeneous applications. Advances in networking combined with needs for high scalability, availability and data integrity caused the creation of complex distributed systems. The integration of legacy systems and inter-operation with remote services substantially increases complexity even further.

Systems often grow to be very heterogeneous because they integrate separate specialized subsystems with intricate data consistency and transactional constraints, spanning multiple domains of authority [BCHG06, AF95, MS]. For instance, consider a dedicated enterprise information system such as SAP, providing customer relationship management (CRM) functions, integrated with a middle-tier application managing orders for the business' online store and Web servers providing the Web front-end for end users. Data for specific business functions, such as processing a catalog-based, personalized order through the online store are distributed across different locations: the enterprise information system, the order system database and a document archive server. Integrating such applications in the presence of different hardware, operating systems, middleware frameworks, networks, programming languages and data formats, with varying levels of performance, load characteristics, reliability and security guarantees becomes very tedious and error-prone. The high degree of heterogeneity mandates a loose coupling of the different applications and integration techniques based on open, widely accepted standards.

The CoCoME (Common Component Modeling Example) case study [RRMP08, DEF+07, BFH+07] illustrates some of these challenges. Figure 1.4 from [DEF+07] depicts a domain model that represents the core entities of the "Point of Sale" (POS) system that is central to the CoCoME. The system is specified in [RRMP08]; it contains elements—such as "Enterprise", "Store", "CashDesk"—that exist one or multiple times in the system and that need to be coordinated consistently. Dependencies between these elements are complex in areas such as control, data, governance, access, and connectivity.

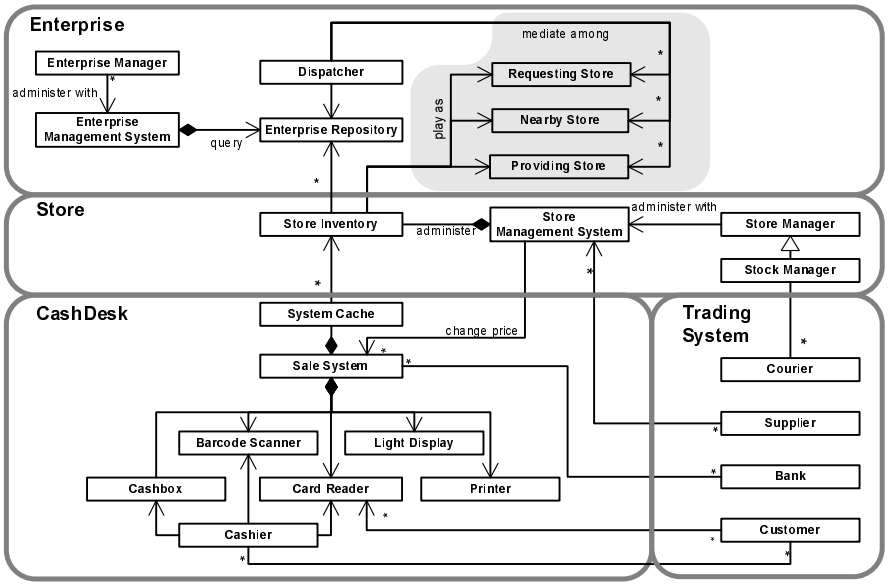

Figure 1.4: CoCoME system roles domain model, from [DEF+07]

Figure 1.5 from [DEF+07] depicts information entities that are managed by the POS system. One particular challenge of distributed information systems is their often complex

distributed data state. Distributed system components interactively providing a service to end users need to access and manipulate system information consistently in a coordinated way. Policy enforcement and access control needs to be applied consistently throughout the system. The failure of a system element or a temporary network outage must not cause inaccurate sales and funds transfer records—not necessarily immediately but eventually after failures have been resolved and global state has been reconciled.

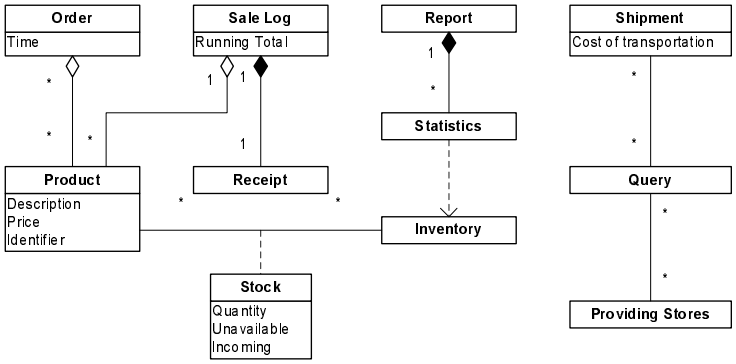

Figure 1.5: CoCoME system data domain model, from [DEF+07]

Common Challenges System components in distributed systems such as the ones described above spread across multiple compute nodes, connected by unreliable networks, potentially spanning multiple domains of authority. Developing distributed systems is of significant difficulty and has received substantial attention in the literature for several decades, cf. [TVS01, Hoa78, Lam78, BS01, CDK05, FK04, And08].

Many reasons exist why distributed systems are difficult to develop. Strategies to enhance resilience to component failures, for comprehensively testing the system, and for maintaining system wide information consistency are complex and not easy to apply. Other challenges arise from the need for particularly safe, real-time, secure and reliable systems, and for maintainable, robust and long lasting system architectures. Many common difficulties in system engineering get amplified for distributed systems: managing changing requirements, describing systems at suitable levels of abstraction with manageable complexity, and managing system evolution over time.

1.1.2 Introducing the Running Example

In the following, we introduce our running example. It is an embedded control system and serves as further example of a complex distributed reactive system with many of the challenges mentioned above. We will use the example to highlight problems and challenges

with developing distributed systems. Subsequently, we will use it to illustrate the development approach proposed in this thesis, demonstrating how it supports system development and addresses the mentioned challenges.

Our case study covers the San Francisco Bay Area Rapid Transit (BART) Advanced Automatic Train Control (AATC) system, a heavy commuter rail train system. The "BART" case study specification document [WKM04] describes parts of the AATC system that control speed and acceleration of the trains. It leaves out other system components, such as communication error recovery and train routing for reasons of brevity. The case study provides a relevant level of detail and shows the complexity and interdependencies of the entire system, yet remains of manageable size. The case study was previously used to showcase the application of formal methods in distributed system design, see [KM04a], enabling a comparison of different methods.

The BART system automatically controls over 50 trains with limited manual operations in cases of emergency and malfunction. The AATC system controls the train movement, optimizing train throughput while maintaining passenger safety. It operates computers at the train stations; each controlling a part of the track network. Stations communicate with the trains via a radio network and are connected to neighboring stations. Each train has two AATC controllers on board, one designated as the master. Trains receive acceleration and brake commands from the station computers via the radio network and feed back train speed and engine status information. The radio network tracks train positions. The system operates in 1/2 second cycles. In each cycle the station control computers receive train information, compute commands for all trains under their control and forward these commands to the trains. All commands are timestamped and become invalid after 2 seconds. Trains go into emergency braking if they do not receive a valid command within 2 seconds. The control algorithm must reflect communication delay, track information and train status, and must never compute new commands that violate the specified safety conditions. To ensure this, each station computer is attached to an independent safety control computer (VSC) that validates all computed commands for conformance with the safety conditions.

Figure 1.6 provides a simplified domain model representation of the BART AATC system, designated the "System under development" in the lower left, and includes some connected external systems. The diagram's notation—explained later in this thesis—is based on a UML [OMG11b] class diagram, depicting components of the system and its environment, as well as various types of relations between these components. This domain model was informed by the specification document [KM04a], but in practice could have been the resulting artifact of a domain analysis activity. It expresses knowledge about the system-of-interest in the context of its environment.

The BART AATC case study is a good exemplar for the system class we investigate, because it touches many aspects relevant in distributed system development. The AATC system is physically distributed with a large number of components with reactive behavior, offering a multitude of functions and operational modes. This leads to a large global

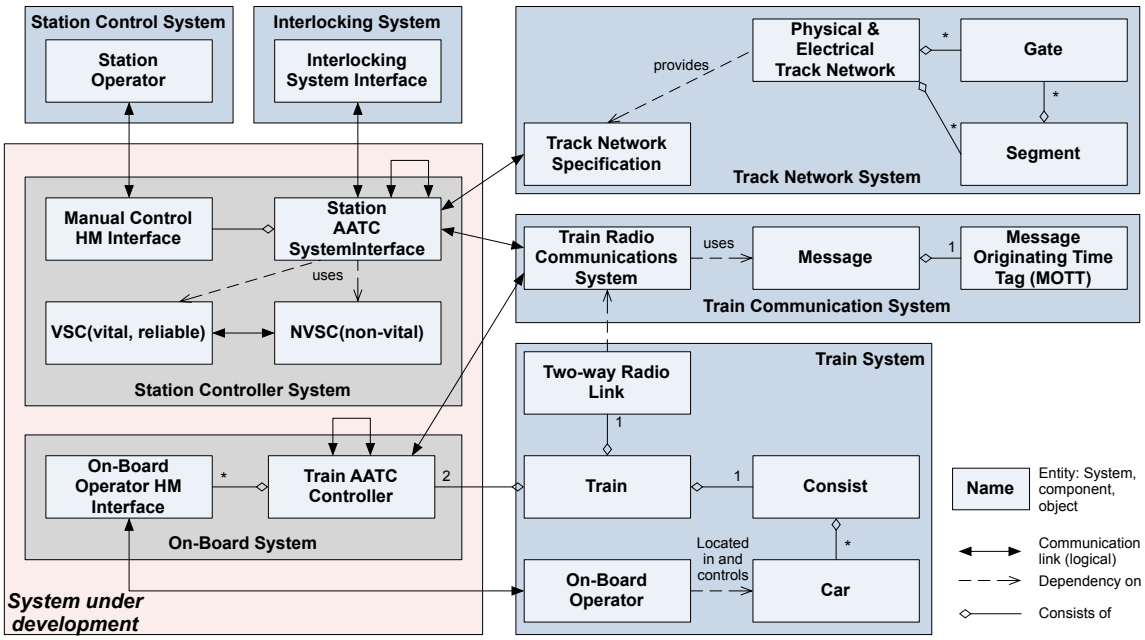

Figure 1.6: BART AATC simplified domain model

state space, with complex interactions, making development difficult. The imposed safety constraints require correct design and implementation, and robustness to failure.

1.1.3 The Importance of System Architecture

System architecture is critical for the quality and success of any distributed system, cf. [GS94, SG96, BCK03, CBB+10, Arc00]. We distinguish the system architecture, the structural organization of the system, from detailed designs that are supported by such an architecture and may change as the system evolves. When developing a distributed system, quality properties such as robustness to failure, manageability, maintainability, security, reliability, scalability and performance need to be designed *into* the system by choosing a suitable system architecture. This activity becomes more difficult when requirements and project constraints restrict the set of possible system architectures.

The scalability of larger-scale systems is determined significantly by their system architecture. System architecture also affects the complexity and efficiency of development. An architecture that combines modular elements, for instance components, is easier to implement and maintain. It is very beneficial to further break down complexity, if architectures are hierarchically structured: breaking the system into components, and the components into smaller components, recursively, as needed. Development work can be performed in parallel at the component level. Architectural elements can be tested individually and reused multiple times.

The documentation of a system architecture is relevant [CBB+10, Arc00]. Documentation needs to be accessible, consistent, well structured, and facilitate communication and knowledge transfer. This supports to the maintenance of systems over long life spans. The system architecture documentation should serve as a living communication artifact for all stakeholders of a system: designers, implementers, operators, but also end users, sponsors, technical writers, customer support, and project management.

We will describe the importance of system architecture in more detail in Section 2.2.2 and subsequently when introducing our development approach.

1.1.4 System Development Approaches and Techniques

In the following, we describe some successful techniques for managing distributed system complexity and development; we also show some of their shortcomings. Neither provides a comprehensive solution for today's challenges with distributed systems.

Component-Oriented Development Software components are well studied in the literature and applied widely in practice. Components leverage the principle of modularity as described by Parnas [Par79], providing beneficial properties such as isolation and reusability. Components serve as modular building blocks of system architectures, cf. [Szy02, BHB+03, TMA+96, Rau02], and are successfully applied in both the academic and commercial worlds. Although the term *software component* is not uniformly defined in the literature, cf. [BHB+03], it can be commonly interpreted as a modular unit of a system during development and operation. Components encapsulate distinct pieces of system functionality that can be designed, implemented, tested, deployed and modified somewhat independently. The architecture of a component-based system describes the construction of the system out of components, and documents their purpose, interfaces, dependencies and other properties.

Component-oriented development methods treat components as primary system building blocks. In distributed systems, components may represent the communicating system elements. The deployment, configuration and interaction of components is often supported by component "middleware", such as CORBA [OHE97] and Enterprise Java Beans [SUN06]. Components react to input from their environment—users, external systems and other components—relayed via a communication medium. If components interact in an environment with direct, immediate and lossless communication, powerful optimizations can be applied to their implementation, integration and verification. In particular, if components are invoked synchronously, i.e. they execute within the control flow of a calling component, typical problems of distributed system design, such as state space explosion during verification, can be avoided. Lossless, direct interaction of components is a degenerate case of the interaction of distributed components in imperfect networks, cf. [Day08]. In the remainder of this thesis, we consider the distributed case unless otherwise stated.

Developing a component-based system requires a matching design. Figure 1.7 shows a flow of development activities and intermediate artifacts, resulting in a component-based system implementation. Development starts in this example with use cases that capture system requirements. If needed for later verification purposes, a (formal) system specification expresses the core properties of the system. Software architects develop a component architecture that supports the functional and quality characteristics of the requirements and/or specification; the architecture represents the structural core of the system design. Designs for individual components, modules, data flow and other system elements are developed as needed. Subsequently, components are implemented, verified and integrated according to the design, resulting in a deployable system. If needed, iterative and incremental development may be applied, potentially with large consequences on the effort required to rework intermediate artifacts.

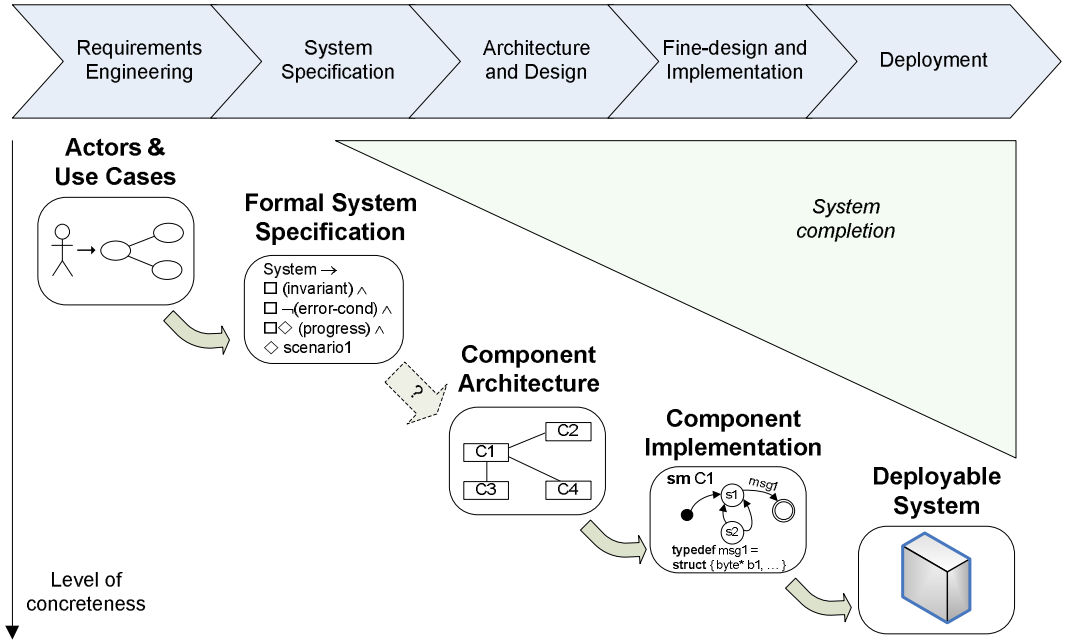

Figure 1.7: Exemplar component-oriented development approach

For development projects in practice, process frameworks such as the V-Modell XT [RBTK05] and the Unified Process [JBR99] provide support for component-oriented development. We will describe existing processes in more detail below.

The described approach has a significant weakness, as indicated by the dashed arrow in Figure 1.7. System architecture development and component design directly build on system requirements or specification; it is a significant leap from the abstract problem domain to a level quite detailed within the solution domain, causing a substantial conceptual gap [RC95]. Bridging this gap requires highly skilled software architects and often many iterations within an iterative development process to realize systems of the desired quality

that fulfill the given requirements and constraints. This transition is not systematic nor repeatable and thus does not constitute "seamless" software engineering. The quality of the outcome and the duration of required development steps cannot be accurately predicted.

Characterizing System Functions A systematic, repeatable work step of structuring the usage functionality of a system—the *services*, independently from the system's component architecture and deployment design, could reduce the conceptual gap between problem and solution space. A structured description of the system's usage functionality—a functional architecture—can serve as baseline for designing an implementation and deployment environment targeted component architecture, which can subsequently evolve for the same functional architecture. We call this design step *functional architecture design* or *service architecture design*. Figure 1.8 shows an enhanced flow of development activities, now applying a seamless approach that develops a functional architecture before developing the component architecture. We will discuss this in detail in the context of service-oriented development, below.

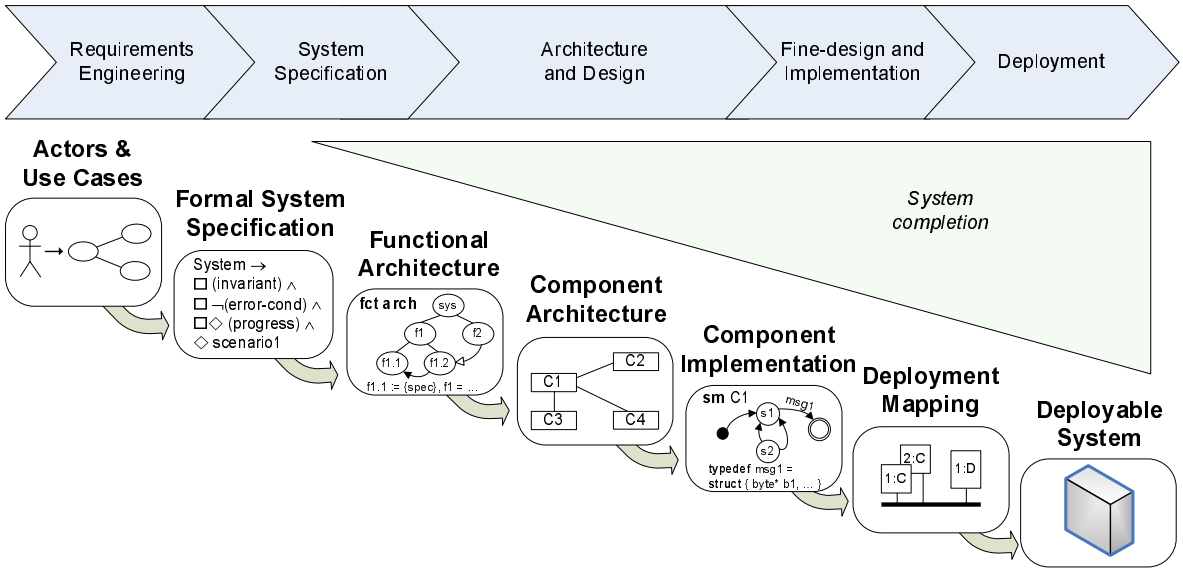

Figure 1.8: Exemplar seamless development approach

Another potential challenge in component-oriented distributed system development is an often tight coupling and intermingling of functional design with design choices motivated by technology. Existing implementation technologies, deployment platforms and target system configurations influence the specific design of components. Technical details, such as specific forms of error handling, communication infrastructure, transaction processing, security etc. mix with functionality, thus forming complex monolithic blocks that are difficult to verify, integrate and maintain. The often applied practice of developing such components independently first, and integrating them later, causes high complexity, which

only increases with larger-scale systems; more and more effort is required to maintain the same level of quality. The tight coupling of functionality and technical design choices also increases the difficulty of changing the deployment structure without changing the functional design of the system, for instance, when migrating from one deployment architecture to a successor in the next release of a system.

A strategy to avoid this tight coupling is to separate the design of components from the description of their deployment in specific target environments. Figure 1.8 depicts a flow that not only separates functional from component architecture, but also defines a deployment mapping for the implemented components. This explicit mapping specifies the instantiation and configuration of the components; thereby targeting a component architecture to specific communication infrastructures, middleware layers and deployment configurations.

Brooks stated the following in 1987 about the refinement of requirements:

> "The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later" [Bro87].

The standing challenge in software engineering is to provide a systematic process to develop systems with sustainable system architectures based on a set of precisely defined, consistent requirements that cover all aspects of the system: interfaces to users and other systems, usage functionality, and quality properties. Such a process needs to cover all development activities, from requirements engineering to deployment, and allow for an iterative refinement of previously created results. The process shall enable seamless and systematic engineering across all activities, keep system services separate from logical system design, and separate logical design from technical characteristics and deployment concerns.

Model-Based Development Model-based development (MBD) or model-driven development (MDD) is a paradigm for systematic software engineering relying on design models as abstract descriptions of systems and their various concerns, cf. [Sch06c, Sel03, Bro05c, SPHP02]. According to [Sta73, BR07, Küh05], each model is based on an original, only reflects a (relevant) selection of the original's properties, and needs to be of use in place of the original with respect to some purpose.

Figure 1.9 shows a model specifying a system. The model serves as an abstraction of the system's implementation. The model itself may consist of multiple views that provide specialized representations of a specific part of the system. Views can range from highly abstract to very concrete, implementation oriented; views are models themselves. A metamodel may exist, defining all model elements and their relationships as well as applicable constraints.

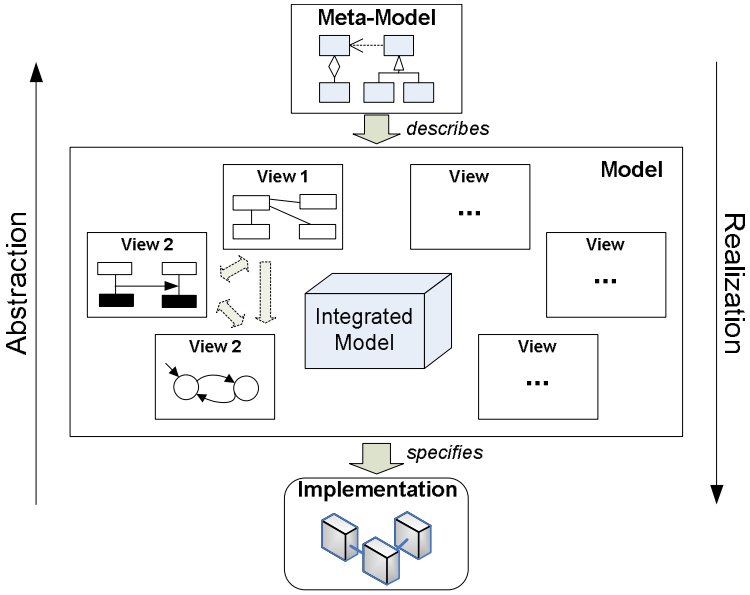

Figure 1.9: Using models to abstract from implementation

Views contain structural, functional, behavioral, operational, data-flow, deployment, and other descriptions, cf. [CBB+10, HKN+07, Kru00a, Arc00]. Abstract structural views of system and domain concepts are sometimes called *domain models* [Eva03]. In order to keep development complexity manageable with ever growing systems, models need to become more abstract, requiring new and more abstract types of views. New views, however, lead to an increased burden of maintaining consistency across all views of the model. An *integrated model* [Sch06a] is a model that represents the same system elements consistently across different model views. A prerequisite for an integrated model is the existence of a well-defined metamodel that defines and constrains the model and describes how its constituent parts relate to one another.

A common source of complexity in system descriptions is the mix of functional design and implementation detail. A strategy for avoiding this situation is to separate functional and implementation perspectives into different related models. An integrity constraint in an integrated model, can for instance require any implementation model to be a refinement of a logical model, respecting the refinement properties defined by the metamodel. Figure 1.10 illustrates this strategy with models derived from our running example. The upper model represents a "logical" model; it specifies functional components connected via directed communication channels. The middle compartment represents an "implementation" model; it depicts implemented components that interact in different ways. They either communicate via a reliable distributed message passing system, depicted as full arrows, or by using in memory inter-process communication, depicted as dashed arrows. This model is a refinement of the upper model, maintaining the behavior interface and communication dependencies of the functional components. The coloring indicates the mapping of functional components to implementation components. This exemplar refinement per-

mits mapping one logical component to multiple implementation components, as well as multiple logical components to one implementation components.

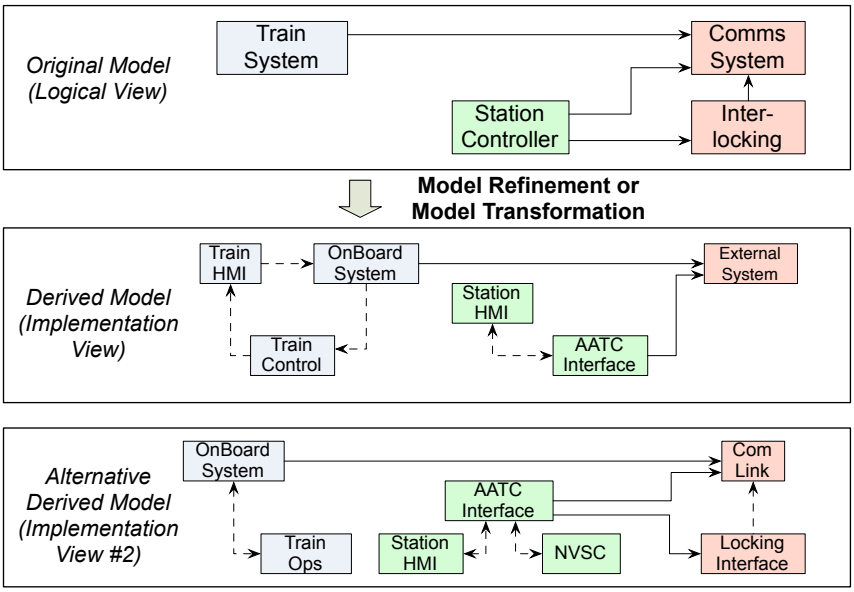

Figure 1.10: Model refinement: from a logical to alternative implementation models

The separation of the logical model from implementation concerns enables a flexible change of deployment design, by developing an alternative implementation model for the same logical model. The lower compartment of Figure 1.10 illustrates such an alternative implementation that respects the behavior and communication interface of the original model. In a situation with two or more candidate implementation architectures, model enactment, comparative prototyping, and other strategies can be applied to select the optimal design.

In general, for a given system, a "logical" model can be developed that is independent of most implementation design choices. This model can be evolved as needed, for instance as requirements change. Implementation choices and deployment configurations can be represented and maintained as explicit mappings within independently managed derivations of the original model. Multiple such "implementation" models can exist for different target environments that all refer to the same original model. The logical model and any derived models must be consistent, according to the rules specified in the metamodel. If the logical model changes, all derived "implementation" models need to be revisited to check for consistency with the logical model. Depending on the rigor of the used model language and the model constraints, some or all of the necessary "refactoring" steps can be performed automatically by supporting tools.

It is possible to provide transformations from the logical model to a derived model. Model-driven architecture (MDA) [OMG03a] and architecture-centric software development [OMG11b]

can provide such development automation. On the downside, the development of such transformations often represents a sizable engineering effort in itself; furthermore, these transformations may not be reusable across projects, limiting their value.

A clean separation of abstract logical models and their concrete implementation models is often difficult to achieve. For instance, certain performance and security constraints might demand specific designs in both models that are co-dependent. This dependency limits the freedom of a purely functionally driven system decomposition. Figure 1.11 illustrates this, showing how three nonfunctional requirements spread differently across models and model entities. This can lead to an undesired tight coupling of abstract and concrete models, making a model-based approach less effective and more costly when model changes are required, for instance to develop alternative implementations. In particular, this is the case for model-driven approaches that rely on providing sophisticated hand-crafted transformations between the two types of models, such as MDA [OMG03a].

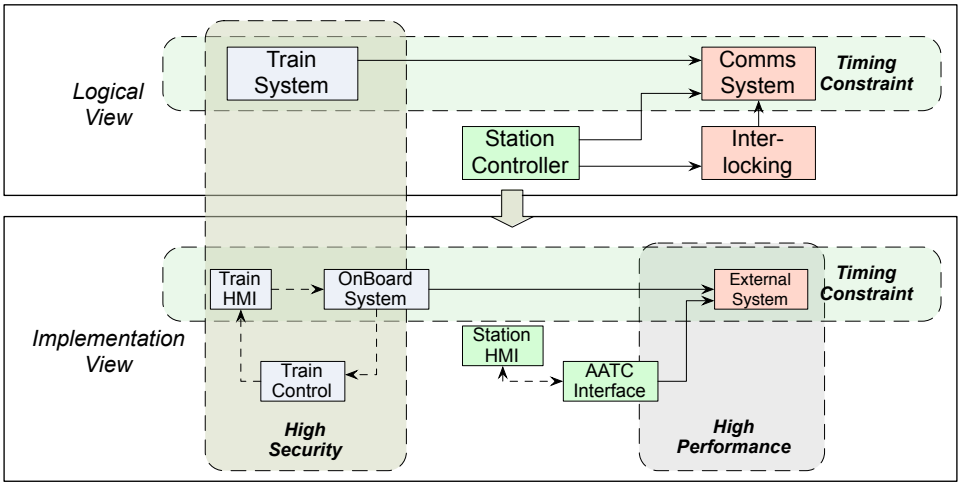

Figure 1.11: Quality-of-service requirements cutting across models

An alternative strategy is to separate all different perspectives of a system design into separate model views that can subsequently be related to derive implementation artifacts. For instance, the design model might comprise (1) a logical model of components and their connections, (2) a deployment model specifying all compute nodes and their communication relations, and (3) a mapping table from logical components to compute nodes. In this case, the information within the design model is sufficient to generate configured deployable compute node processes from the design model. This is the strategy that we will follow for the development process introduced in this thesis. It is similar to the approach described in [KMM06].

In this thesis, we are particularly interested in model-based development approaches for component-oriented systems. We argue that components provide a robust abstraction and

unit of encapsulation as building blocks of system development. We advocate for integrated model-based development processes that provide systematic and relatively seamless transitions through the abstraction levels of system development, from requirements engineering to deployment, separating logical from implementation concerns.

Multifunctional System Development The systems we are interested in are often multifunctional, i.e. they are providing many different functions to their environment. Often, such functions appear to be independent of one another at first glance. On closer analysis, unexpected dependencies may be detected, and the impact of a change on other functions is undetermined. Functions may explicitly depend on other functions, or be combined out of subfunctions. Figure 1.12 illustrates complex functional dependencies that can occur within a multifunctional system; it shows a directed graph of functions with the system at the root. Subfunctions are shown beneath a function. Other functional dependencies are depicted by labeled association arrows between functions, where the label indicates the type of dependency, such as "depends on" and "controls". Functional dependency analysis is a topic of ongoing research in requirements engineering, see [Bro10, GM09, GHH07, Kle06, Rit08, Deu08, Sch08, Gru10]. The desired and undesired dependencies of functions in a system have been extensively studied as "feature interactions" in telecommunications systems cf. [Zav93, Blo97] and in general [Vog15].

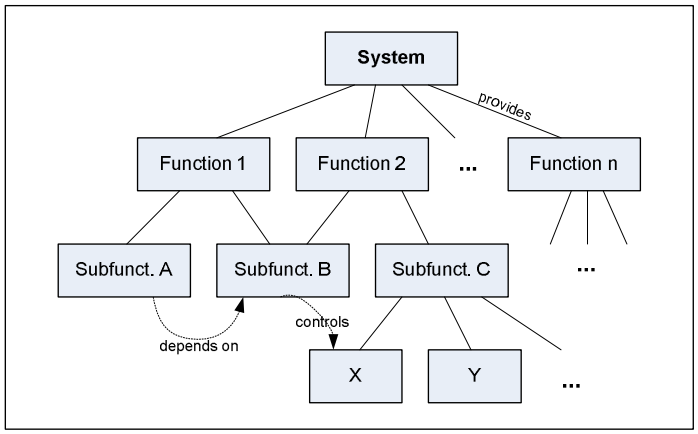

Figure 1.12: Function dependency graph for a multifunctional system

For telecommunications systems, for instance, [Zav01] defines a feature as an "optional or incremental unit of functionality". Typical examples include "Call forwarding" (CF), "Originating Call Screening" (OCS), and "Call ID" (CID). Features can be added to a core system and can be developed independently. A feature specification contains an action, enabling condition, and priority. The action is performed, if the enabling condition is true and the priority is the highest. A feature-oriented description is a "description of a software system organized by features, consisting of a base description and feature modules, each of which describes a separate feature" [Zav01, ZJ01].

Multifunctional distributed reactive systems consist of multiple interacting components; most system functions are provided through a collaboration among these components. We speak of functionality crosscutting the system structure. This crosscutting nature of functions is visible throughout abstraction levels in design models; a functional dependency on the logical level corresponds to a comparable interaction on the implementation level. Figure 1.13 illustrates how three functions crosscut the components in the logical model and in the implementation model. The curved lines sketch flows of information and control; these flows originate from the environment and crosscut several components in the model, indicating interaction, and end with some result visible to the environment. The "Emergency Signal" function, for instance, is triggered by the train operator or a passenger on the train interacting with the Train HMI (Human-Machine Interface). Via the train's OnBoard System, the signal is relayed through the BART Intercom system, potentially leading to an emergency break command to the train. The "Operator Control" function shows a train operator manually operating the train through the Train HMI, leading to commands to the Train Control via the OnBoard System. Resulting status information is provided back to the operator as feedback.

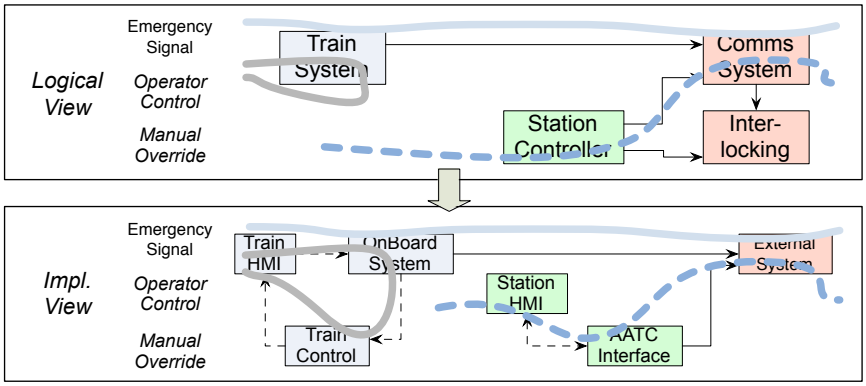

Figure 1.13: Functions crosscutting entities in logical and implementation models

Nonfunctional properties associated with the functions, such as security and performance, may similarly affect all design models. Effective model-based system development approaches need to separate logical and implementation details cleanly within models that are systematically related across abstraction levels. Ideally, specifications on the logical level can be related—unchanged—to all implementation models and target architectures.

Service-Oriented Software Development Separation of the functions of a system from its component architecture decouples system behavior from implementation details. This is particularly true for multifunctional systems as described above. A promising approach in this direction is *service-orientation*; it gained significant popularity over the last decade in academia as well as in industry. Services and service-oriented methods are widespread

in many application domains, cf. [SD05, AH06, SH05, TRH+04] and have their roots in the feature-oriented approaches for telecommunications systems [Fre04, Zav93].

At the core of any service-oriented approach is typically a concept of a *service*—representing a function or feature—as first-class entity, rather than a set of components implementing the function. An implemented service will be provided by one or several of the system's components; nonetheless, a consumer of this service only depends on the presence of this service and its interface in the system, not on the specific component providing this service. Creating service-oriented software systems becomes more of a task of arranging and composing functions in a way most suited for a given set of requirements, rather than designing components and integrating them into systems. Defining services through their observable interaction behavior is a good way of characterizing system behavior without constraining how system components actual work inside. This deemphasizes implementation details when designing application level architectures up to a point where components get fully abstracted away, leaving a purely logical arrangement of services and their dependencies. The level of abstraction in service-oriented system designs is therefore higher than in comparable component-based designs. When applied properly, this approach leads to more robust, maintainable, loosely coupled systems that can flexibly adapt to changes to target environments and deployment configurations; in some cases without even restarting the system.

Figure 1.14 illustrates how services can provide an interface to a layer of "domain objects", for instance a set of existing business components. Domain objects implement the functional logic of the system and provide persistence and transactional behavior. Each object has a distinct responsibility in the system; some objects may depend on others. In many larger systems, domain objects often using different technologies exist that are cumbersome to maintain, extend and integrate, for instance to add new system-wide functionality. Services can assist in the structuring and integration of such systems. Services—represented as entities on the figure's service layer—provide the system's functionality to the environment. Services represent well-defined, independent functions within the system. Services may interact with other services; complex functions are often realized as an interplay of multiple services. Together, when implemented, the services form a service-oriented architecture. Each service governs a number of domain objects [Eva03]. The service thus acts as an external interface to a cluster of connected domain objects, following the *Façade* design pattern [GHJV95]. It shields the domain objects from direct access by the outside world; this contributes to a loose coupling of principal parts of the system and also allows for an easier extensibility of system functionality. This pattern can be applied, where complex, often distributed applications are expected to offer externally accessible interfaces.

The existence of a service layer abstracting from more fine-grained domain objects showcases the importance of functions as primary driver to structure the system design. Designing system behavior by identifying and connecting a set of well-defined services, and structuring the system architecture accordingly, results in loosely coupled, flexible systems that remain manageable and maintainable, even when becoming larger-sized and more

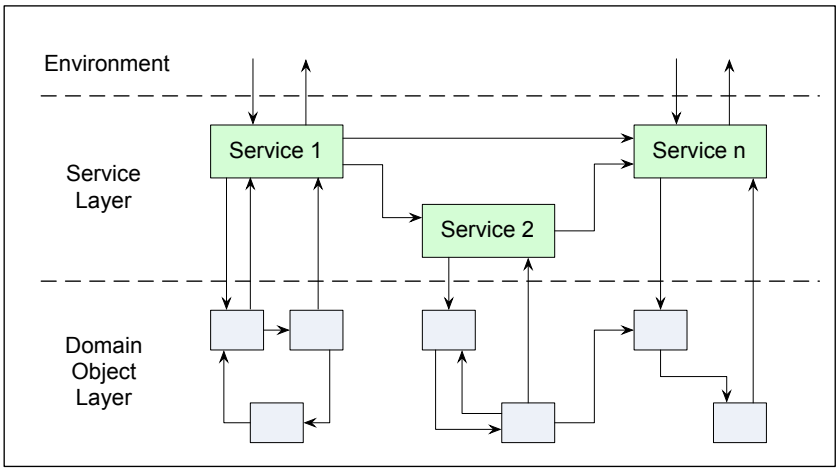

Figure 1.14: Services providing a façade to a layer of domain objects

heterogeneous, such as when integrating them with other independent systems.

Systems that are engineered to provide services are called *Service-Oriented Architectures (SOA)* [Bar06]. Applications on such systems are primarily built from services; they require supporting infrastructure to configure and deploy the services in the deployment environment, to make them discoverable by consumers, to allow interoperation with remote services, and to facilitate service orchestration and composition. The term service-oriented architecture does not imply any specific technology or network infrastructure. The most prevalent and well-known implementation style for a service-oriented architecture is based on the Web Services Architecture [W3C04b], applying technologies and standards from the domain of Internet and World-Wide-Web applications, such as WSDL for describing service interfaces, SOAP for message encoding, and HTTP/HTTPS for message transport [New02]. The services in this style are simply called Web Services [STK02] or WS-* services. Architectures based on HTTP and light-weight REST services, see Section 2.3.4, are omnipresent in today's Web environment, providing flexible, scalable interoperability. SOAs resting on message passing middleware are gaining momentum; they provide services by sending and receiving messages.

The OOI Integrated Observatory Network, for instance, is a large distributed software-intensive system based on a service-oriented architecture [OOI14]. It provides services built on a message-based communication infrastructure, using the AMQP protocol as the transport layer [OOI14]. Figure 1.15 depicts its core set of services. Colors group services by larger area of responsibility, such as data management. Arrows indicate service dependencies. Each service has a different functional responsibility in the system, ranging from "low-level" identity management, to "higher level" science data and sensor (instrument) management. Each service provides between 10 and 200 "operations" as distinct functions that fall within the area of responsibility of the service. Services are designed in layers,

with higher level services and their operations only using lower level services to provide their functions.

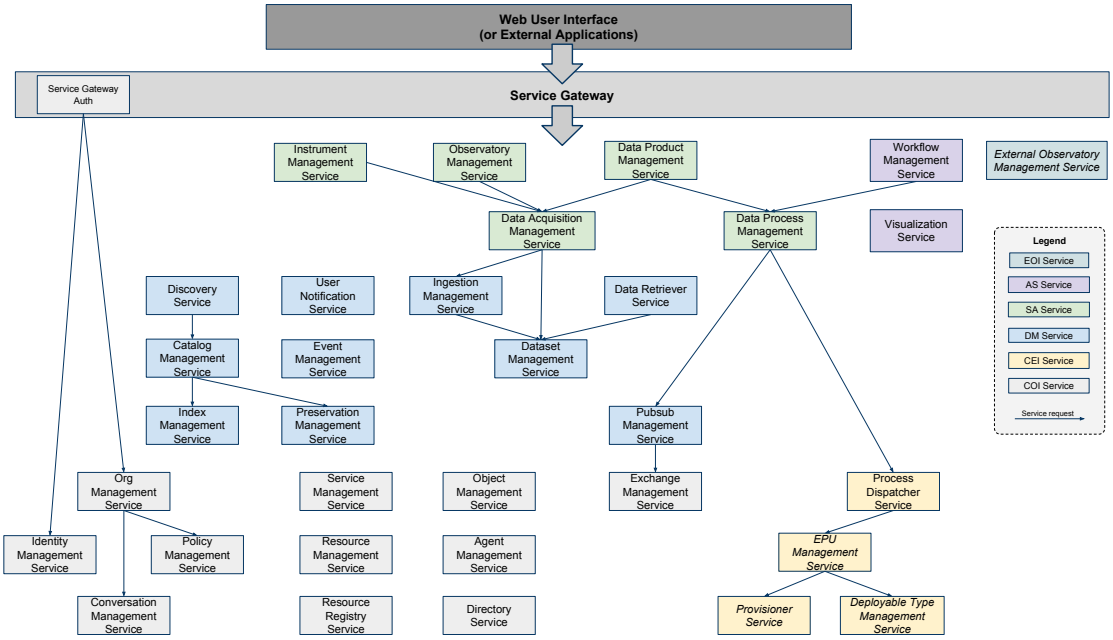

Figure 1.15: OOI Integrated Observatory Network services

An architectural pattern that has gained substantial traction in the recent years is *Microservices* [New15]. This pattern propagates systems comprised of relatively small, independent services built around central elements of the business domain. Each service provides its own deployment and persistence strategy, and there is no need for a shared infrastructure such as an Enterprise Service Bus. Each service is essentially its own separately deployed component. This provides for a maximum decoupling of system elements, particularly useful for larger systems, and for simplified system deployment and maintenance processes. Component container technologies such as Docker[1] make it feasible to quickly deploy and upgrade such independent service components.

Service-oriented system architectures, and to a lesser degree service-oriented approaches to system development, are prominent in business information systems with Web Services, Web-based applications, and in the telecommunications domain [Zav01]. They are emerging in the automotive domain [BK04, BKM06, BKM08] and in other application domains. Despite the attention that service-oriented software development receives, it appears from our research that precise, scalable, domain-independent development approaches and suitable development processes are still missing, or provide only a partial solution to the challenges described above.

[1] `https://www.docker.com/`

In the absence of a commonly accepted definition, we generalize a notion of service that encompasses most existing definitions, as expressed by the following definition:

> *A service represents a function of an application, described by an input-output interface, coordinating implementation level entities.*

We will apply this notion when discussing service-orientation and the benefits of services in more detail in Chapter 2. In Section 3.2, we will introduce a formalization of a service based on the FOCUS formalism, consistent with the service notion above. This formal definition will provide a precise interpretation for a service as a generalization of a distributed component in a system architecture, specified as a partial function over infinite streams, describing the syntactic and behavioral interfaces of the service. A service describes an independent "function of an application" as a partial behavior that can be combined with other such services. Subsequently, all services can be mapped to a set of distributed components of the system that specify behavior for all inputs, ready to be implemented. Thereby, the services "coordinate" components as "implementation level entities". Service interfaces will be part of the public interface of a system, thereby being "described by an input-output interface".

We believe that only a precise, formal definition of a service can support a robust system engineering approach that spans all main activities from requirements engineering to system operation, with services as first class entities, delineating them from related concepts such as requirements, use cases, and components. The formalization is required to provide precise interpretation for the elements and steps of an approach, thereby supporting automation by tools and verification of correctness of changes to the system design. We will show that our formalization does not substantially restrict the applicability of a service-oriented approach, supporting a domain independent process that is flexible for extension.

Software and System Development Processes Developing large-scale software systems is cumbersome and requires the application of good engineering and management practices over a prolonged period of time, often involving multiple organizations spread across multiple locations. The formalization of best practices and effective methods in a repeatable way led to the definition of today's software development processes. A development process governs and drives the development life-cycle of systems and software. A development process defines activities, work products and roles that are responsible for carrying out the activities that manipulate work products. It breaks down system development into smaller activities, which can be carried out systematically and repeatedly. Core engineering activities include requirements management, design, implementation, integration and testing. In many development processes, the engineering activities are complemented by project management, quality assurance and configuration control activities.

Goals of applying a development process include increasing the productivity of the development activities, the quality of the results, and the predictability and repeatability of the outcome. Therefore, the process improves manageability of software and system development projects, and provides the foundation for systematic system engineering.

Development processes come in many flavors. They range from very abstract, providing only requirements on specific processes without prescribing any activities or work products, for instance ISO 15288 [ISO08c] and CMMI [CMM06], to very concrete, detailed domain specific guidance that often suggests the use of specific tools and methodology. Plan-driven processes emphasize management, planning and documentation activities, while agile methods provide light-weight processes that directly support iterative software development. In general, more abstract development processes are also more generic and provide less guidance and support.

The artifacts—instances of work products—created and manipulated by a development process represent its main outcome. Artifacts include plans, reports, documentation, software designs, and developed software components. There exist dependencies of many different nature between artifacts. A core engineering dependency, for instance, requires traceability of system requirements to design artifacts and to the actual developed system components. Consistency rules demand that every requirement is compliant with the constraints of the project, is non-contradictory, testable, represented in the system design, implemented within one or multiple components, and verified in design and implementation. A substantial number of dependencies and consistency conditions between artifacts contributes to what is commonly called "requirements traceability".

The basis for asserting consistency between development artifacts is a precise artifact model that characterizes all artifacts and defines all dependencies between artifacts. Such an artifact model is often dependent on the applied development strategy and the specific development approach.

We see strong value in the combination of a service-oriented development approach with a systematic development process, in particular when applied to the complex domain of software-intensive distributed reactive systems. However, there exists no sufficiently precise artifact model for the service-oriented development of distributed reactive systems in the literature. We will provide references to existing work in Section 2.1.

1.2 Key Properties of System Development Approaches

We can distill a set of key properties of development approaches for multifunctional distributed reactive systems from the analyses discussed above. These properties represent what we perceive as the essence of effective system development. Current development approaches do not satisfy enough of these properties to support continued growth in system size at equal or higher quality, and at equal or lower cost.

The following sections briefly explain each property and discuss how its presence benefits, respectively its absence negatively affects, system development. We mention current practice and existing shortcomings.

Logical and abstract design models This property characterizes the existence of such models, which describe parts of the system design on a higher level of abstraction; they are important to manage complexities that exist in larger systems with many distributed components and operational states. Often, too many functions and quality properties are spread across implementation oriented models. The absence of such models leads to several problems: Verifying the functional correctness of the design can be difficult and component specifications get too complex. Integration of components becomes cumbersome, because of unspecified dependencies caused by crosscutting functionality. Maintenance and migration of systems is difficult, because functionality is closely entangled with characteristics of the target environment; design models become obsolete when migrating to new architectures. Most of today's development approaches focus on designing component-based implementations for specific technologies, middleware platforms, and deployment environments, such as for Web Services and J2EE frameworks. More abstract specifications of the logical system architecture are often absent. Functionality of the system is entangled with technical characteristics of the target deployment environment and with the actual deployment configuration of the system.

Functional architecture Specifying the individual functions—the services—of a distributed multifunctional system and their dependencies is important to organize the usage view of a system and to detect any unwanted functional interactions early. This explicit management of functions enables the addition, removal, and change of functions as the design progresses. The lack of such a functional architecture, and a general lack of managing system functions, can lead to unmanageable distributed system state and interaction behavior; this occurs in particular for crosscutting functionalities, such as failure management and security. Unwanted feature interactions will be detected late during integration activities, leading to reduced productivity and potentially lower system quality. Very few design approaches currently manage system functions explicitly.

Adequate design models Design model techniques that are suitable for the application domain, effective to use, intuitively comprehensible, non-redundant, and with a low barrier of entry are key to designing larger-scale distributed systems. The resulting models enable the designer to precisely express desired system properties, and can serve as tangible artifacts, communicating system design intent that can readily be verified. Properties include functional, safety, progress, performance, and security properties. Methodology needs to exist to relate overlapping models, assisted by development processes and tool support. Without adequate models, system design becomes cumbersome and development productivity decreases, due to design inconsistencies detected late and integration related

problems. Many effective description techniques exist in the literature and get applied in practice. Often, it is an unclear formal interpretation of these models that is holding back more advanced uses, such as seamless tool support.

Formal system model A formal model provides a theory for the structure and behavior of systems and their components. System behavior and desired system properties can be expressed with respect to such a model. Furthermore, the interpretation of design models and model transformations can be expressed. We speak of the formal model as the "semantics domain". Without such a formal system model, the meaning of design models may remain ambiguous and imprecision in design intent results, with negative consequences as described. Formal verification of designs becomes difficult and error-prone. Few formal system models exist that are suitable for multifunctional distributed reactive systems. If they exist, they may exist as independent theories with limited applicability.

Domain independence A domain independent development approach provides guidance for distributed system development from multiple application domains. The advantage of a domain independent approach is its wider applicability and thereby a larger user community to prove its viability and enhance the approach. A larger market often goes along with better tooling. An approach specialized to a single domain bears the risk that fewer best practice solutions to general distributed system problems got integrated, because of the time and effort it takes to maintain a development approach and keep it current. Few application domain independent development approaches exist that provide core answers to the challenges of developing multifunctional distributed reactive systems. Where existing, they require mechanisms for adaptation to specific application domains.

Seamless development System development works best if there are few conceptual gaps between development artifacts, and if the steps to refine a design are small and repeatable. A development process that supports all activities of system development equally, with little room for surprise, will lead to higher predictability of result quality and cost, and higher overall development productivity. If conceptual gaps exist, then system designers must add their own interpretations and design decisions, resulting in unclear and unstable designs. Inevitable revisions and communications loops can dramatically bring down a development team's productivity. Nonetheless, few existing development approaches address the conceptual gap between system requirements and design, for instance.

Iterative development It is very hard to design and build a system in one shot, in a "grand design". An iterative approach enables a repeated application of design refinement and revision steps. Each iteration can stabilize the design, fix any discovered issues, and add still necessary system functions. Sequential "waterfall style" succession of the development activities rarely produces results of desired quality and cost for larger scale software

systems, even with effective change management in place. Iterative processes are the norm rather than the exception for successful software development projects. Such processes must ensure that work on any artifact can be performed incrementally, and that checks are in place to assess and ensure overall artifact consistency.

Compositionality The ability to combine components to more capable components, without losing the original components' properties. If an approach is compositional, then components can be verified in isolation. A composed system will retain some or all of the components' properties. In addition, the properties proved for a composite specification also hold for a composite implementation, as long as the individual components remain true to their respective specifications. Compositional verification applies these principles. The benefit is scalability of the development approach to larger systems. Approaches and processes that do not provide such scalability through composition will be inefficient for any reasonably sized development project.

Maintainable design models Design models that are too complex, not intuitive, don't support iterative development, or don't link to derived designs and implementations tend to diverge over time and become useless. Design models remain only useful if they are living artifacts during the lifetime of a development project, are used by as many stakeholders as possible, and are maintained in a consistent state. The ability to flexibly modify them, keeping consistency with derived artifacts is an essential requirement of a development approach.

Tool support Automating repeated steps in development is one of the best strategies for making an approach easier to use and more predictable. The absence of process automation and tool support introduces the potential for human error during the development activities and leads to lower development productivity. This applies in particular, when transitioning between artifacts of different type, such as from a logical design model to a fine-design implementation model. Tool support will be less effective with more lenient definitions of process metamodel, and semantics of design notations and refinement operations. Some development approaches cannot be effectively supported by tools, because they are lacking precise definitions of concepts.

Characterized development approach An approach is clearly characterized, if it documents all benefits and shortcomings as applied to a specific system class. Trade-offs, such as more overhead and training required to maintain certain design models in return for more predictable development planning, need to be known or be quantifiable in advance for a given project size. This will make it possible to select or reject an approach. Otherwise uninformed process choices result in less productive development efforts.

Development process A process provides a systematic, repeatable path to developing a system. This limits the variance in project cost and schedule and ensures a certain quality of the development outcome. The absence of a defined process can lead to less productivity and lower system quality. Processes are very common in larger development projects; they are rarely combined though with a precise, formally founded development approach.

Precise process definition A precisely defined process specifies the meaning of its work products, activities and role responsibilities clearly; preferably in form of a process meta-model built on the foundation of a system model or system theory. The more precise a process definition, the easier it gets to automate and support by tools. A precise process is a prerequisite for assessing the completeness, correctness, and consistency of development artifacts, and for assessing the correct completion of development activities. While there are several formally founded development methodologies with notations and refinement operations, there are few similarly well defined development processes.

Summarizing the Key Properties Table 1.1 lists the described key properties. It provides our assessment of the degree of their support for three classes of existing development approaches. We include traditional and model-based component-based approaches as well as service-oriented approaches. The values range from low (– –) to high (++) degree of support. The table also indicates the risk to development success in case a property is not addressed by an approach. We rank the approaches by counting high values as 4 points and lower values respectively less. Traditional component-based approaches rank last, while model-based component-oriented and service-oriented approaches tie at a much higher level. Each class has its own advantages and disadvantages. Component-based approaches are generally more mature, while existing service-oriented approaches address the challenges of large-scale distributed systems better.

The key properties can be used to evaluate development approaches for multifunctional distributed reactive systems. In general, and if applied appropriately, the more properties are fulfilled, the lower the risk of failure or for missing the goals. Additional specific requirements and constraints will come from a project's characteristics, such as the application domain, project size and duration, and the expected stability of functional requirements. It may not be possible to find an existing approach that fulfills all these requirements while fitting the project constraints. Determining the trade-offs when choosing between properties is an important step in selecting a suitable approach. Of course, it is also possible to augment an existing approach with proven methods increasing certain properties, but this has a much higher upfront cost in process design that might only be feasible to larger organizations.

Table 1.1: Development approach key properties

Property	Supported in traditional component-based approaches	Supported in model-based component-based approaches	Supported in service-oriented approaches	Development risk increase if absent
Logical and abstract design models	– –	–	+	high
Functional architecture	– –	– –	+	medium
Adequate design models	–	+	○	high
Formal system model	–	–	–	medium
Domain independence	–	+	○	low
Seamless development	–	○	○	medium
Iterative development	–	○	○	high
Compositionality	○	+	○	medium
Maintainable design models	– –	+	+	high
Tool support	–	+	+	high
Characterized development approach	○	+	○	medium
Development process	–	○	○	high
Precise process definition	–	○	–	medium
Rating Index	12	28	28	

1.3 Problem Statement

As described above, service-oriented methods have proven effective in designing distributed systems by addressing the most prevalent development complexities. These methods shift the focus from the artifacts of system implementation to formalizing the systems' usage: defining the *services* of the system and their *interactions*. They gain further strength when applied as a model-based approach. Table 1.1 substantiates this analysis and shows that service-oriented approaches provide high abstraction and functional design models to keep system complexity manageable. They still have weaknesses in systematic iterative and seamless development.

Based on our research, we conclude that there exists no work in the literature that combines a formally defined service-oriented approach with a systematic development process,

resulting in an approach that is scalable to larger systems and applicable to multiple application domains. Our goal in this thesis is to propose an enhanced approach that addresses all of the above key properties at a level suitable for efficient practical applicability. To this end, we will define a development process and service-oriented development approach, based on service-oriented methods and a formal distributed system theory.

1.4 Contribution

Summary The core contribution of this thesis is the definition of artifact and activity models for a development process for the service-oriented design of software-intensive, multifunctional, distributed reactive systems. This process manages development complexity as well as system complexity caused by component distribution. It incorporates a model-based, service-oriented development method that emphasizes designing a clear functional architecture of the distributed system, comprised of services provided to the environment, subsequently mapped to a hierarchical architecture of distributed components. We specify process work products and integrity constraints in our process artifact model. This includes all requirements documents, specifications, design models and design model views, other intermediary development products, and system implementation artifacts. We define activities and their orchestration in our process activity model, supporting seamless incremental and iterative project execution strategies. Both artifact and activity models are based on a precise process metamodel. We present precise definitions of model elements, notations, and transformations of our approach, using an existing formal model of distributed systems. The process and its artifact model are carefully designed to support the formal model.

We claim that our service-oriented approach addresses the challenges described in the previous section. The proposed combination of systematic engineering process and formally founded development method realizes feasible trade-offs satisfying the described key properties for typical development situations in the system class we consider.

The primary topics with *novel work* presented in this thesis include the service-oriented development process artifact model, the process activity model, the integration of the core process as an extension of the V-Modell XT, the description of a comprehensive tool support architecture, and the implementation of the SODA prototype. These topics will be described in Chapters 4 through 6. Figure 1.17 in the outline section shows the main contributions of this thesis—our approach and its validation—and how they relate to background materials and foundations. Contributions—in darker shading—and relevant input are laid out according to the chapter structure of the thesis.

Figure 1.18, below, highlights important contributions of this work in the context of preceding and concurrent research work. Another valuable outcome of this work is the consistent and comprehensive presentation of a broad array of materials related to service-orientation, including several formal models. We integrate formal work from a variety of sources and

present definitions consistently, used as comprehensive formal foundation for our service-oriented process. In the following, we describe our approach in more detail and highlight novel work.

Description of our Approach Our proposed system design approach cleanly separates functional architecture from implementation details into distinct views within an integrated design model, defined within the process artifact model. Each model view applies notations tailored to the view's level of abstraction. Figure 1.16 depicts parts of the integrated model for our running example, consisting of multiple interrelated views. The "Service View" in the upper left compartment of the figure represents a functional architecture, cf. Figure 1.14. It identifies the services and specifies their dependencies. The "Logical Component View" below shows a component architecture that can offer the specified services. It represents a logical component design with implementable components, potentially composed out of subcomponents. A mapping exists between services and logical components. Further views exist, including a "Deployment View" that specifies deployment choices for the logical component architecture, such as the number of instances for each component, their distribution across compute nodes, as well as their communication relationships reflecting communication middleware for the target environment, such as Web Services. A mapping exists between logical components and deployment elements.

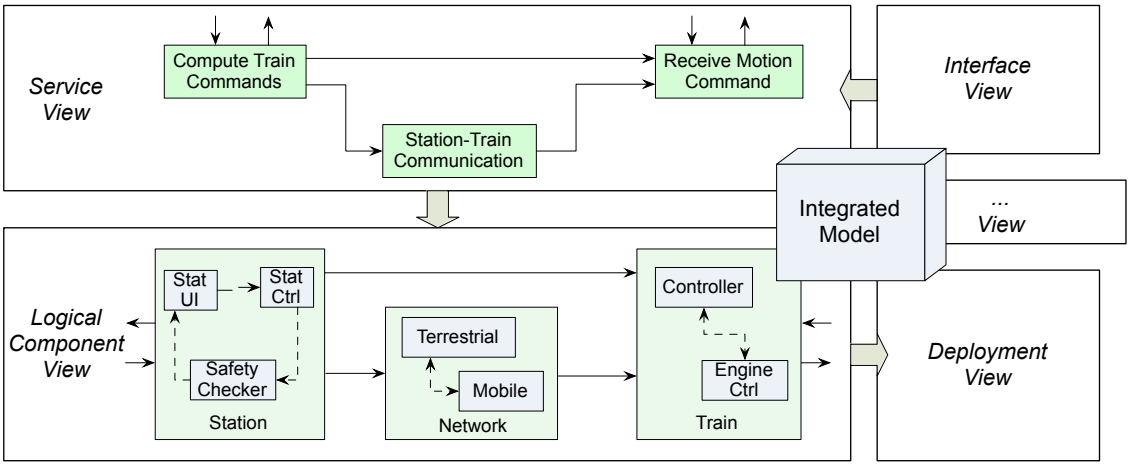

Figure 1.16: Integrated service-oriented architecture model with consistent views

Our service-oriented approach emphasizes functionality as fundamental structuring principle of system design models. Services represent the formalized independent functions that a system or a component offers. Services are related to system use cases; in contrast to these, services provide comprehensive, consistent, and detailed specifications of system behavior at a high level of detail. In particular, services formalize the interactions between system components. Interactions are inevitable in a distributed system and their correctness is

essential to the correctness of the system. Designing these interactions systematically together with system architectures that effectively support them is a critical development activity. Our service notion provides a linking element between system requirements and detailed implementable system designs. Developing systems around their services enables more seamless refinement of requirements into fine-designs and implementations, by narrowing the conceptual gap between problem space and solution space.

Our approach is not limited to one particular application domain. It targets the core challenges of distributed system design: the coordination of distributed components and their interactions. Our approach supports the design of Web Services based information systems as well as of embedded automotive control systems, such as a power windows control unit. By separating functional architecture from implementation design, we are able to maintain mappings to alternative implementation designs for the same functional specification. This eases the evolution of existing architectures, the migration to different technology platforms, and the reuse of proven functional architectures across application domains. The same functional architecture—specified as service interactions—can for instance be implemented as an object-oriented program with local method calls, as a Web Services based distributed Web application, as an application using Remote Procedure Calls (RPC) on a distributed object broker middleware such as CORBA, as a service-oriented system using message passing, and as a system of independent electronic control units (ECUs) connected by a common broadcast bus. Larger heterogeneous system integration projects might even combine multiple deployment technologies into one system of systems.

Our approach is designed around the principles of modularity and composition, keeping it scalable to ever increasing system sizes. It applies a strong notion of interaction interfaces and ensures that interfaces specified on the system level remain true for any design refinement. Compositionality guarantees that properties verified for a composite of component specifications hold true for a system composed of implementations of the specified components, providing the implementations can be successfully verified against their specifications. This means that systems can be composed of components in a bottom up development style, or alternatively, high-level system elements can be decomposed top down into more manageable components.

In order to define the meaning of all elements of our approach precisely, we make use of Focus, a comprehensive theory of distributed systems based on the mathematical notion of streams [BS01]. It provides the semantics domain for the interpretation of concepts, notations, and operations we use or define in our approach. In addition, we make use of a formal development methodology for distributed system components using extended Message Sequence Charts (MSC) [Krü00b]. In this thesis, we present the relevant parts of Focus, the formal system model, and the extended MSC approach coherently. Where required, we extend them carefully with our own definitions, while preserving basic properties of the original theories, resulting in a comprehensive formal model that serves as foundation for all our definitions and subsequent discussions.

The integrated nature of our model-based approach requires that all views for one design model must be kept consistent. Two mechanisms satisfy this requirement in our approach: property-maintaining model transformations and consistency enforced by a process. Model transformations exist that perform consistency preserving model changes. Examples include certain *refinements* that preserve significant model properties while adding design detail, and *refactorings* that restructure designs without impacting system behavior, for instance to rename system elements. As core contribution of this service engineering approach, a well defined process supports the consistent development of the design model and the coordinated orchestration of concurrent and dependent development activities, derived from and unfolding the semantic space provided by the formal model. Process activities, defined in the process activity model, guide the manipulation of system design elements in individual views with exit criteria and analytic steps that keep the overall model consistent throughout the development cycle. For instance, after a designer updates the service view, the process requires a revisit of the component view to make it consistent with the service view. The process also defines checks to assess consistency within and across artifacts, for instance within model views. Similarly, it supports the tracing of requirements throughout all elements of the design model. Process work products and their consistency rules are defined in the process activity model. It also suggests suitable notations. All process activities can be applied iteratively, for instance first resulting in a compact realization of the basic system use cases, subsequently refined into more comprehensive functional architectures and implementable system designs. We define all elements of our process, including artifacts, activities and consistency rules, using a process metamodel. This enables us to reason about artifact completeness and consistency, further enabling provide process automation and tool support.

Demonstration and Evaluation of our Approach To demonstrate the practical applicability of our approach, we outline a process supporting tool architecture and embed our process into an industry standard development process. Our tool architecture supports and automates our model-based development process. It is capable of authoring the development model artifacts, assessing artifact consistency, and applying model transformations. We discuss the architecture and conclude that it realizes an important element of a comprehensive service engineering solution. Furthermore, we combine our approach with the established systems development process model V-Modell XT. We introduce a service-oriented extension of the V-Modell XT, embedding our domain independent service-oriented approach. We provide a correspondence of concepts, relating our generic process and the service-oriented V-Modell XT extension.

We use the BART AATC case study as our running example to demonstrate definitions of our process and its application throughout this thesis. This case study provides an exemplar for systems in various application domains. We use the BART case study to evaluate how our approach accomplishes addressing the above stated challenges in developing distributed systems. Table 1.2 summarizes the benefits of applying our approach when compared to existing traditional, not service-oriented approaches.

Table 1.2: Expected influence of our approach on success criteria

Property	**Degree of support existing approaches**	**Degree of support with our approach**
Logical and abstract design models	– – to –	++
Functional architecture	– –	++
Adequate design models	– to +	++
Formal system model	–	++
Domain independence	– to +	+
Seamless development	– to ○	+
Iterative development	– to ○	+
Compositionality	○ to +	++
Maintainable design models	– – to +	+
Tool support	– to +	+
Characterized development approach	○ to +	+
Development process	– to ○	++
Precise process definition	– to ○	++

We conclude from this evaluation that applying our process leads to a more comprehensive understanding of the system under development, which positively influences system quality, and limits the variability of development duration and cost.

1.5 Outline

The remainder of this thesis is structured along the following outline, depicted in Figure 1.17:

In Chapter 2, we describe how service-oriented development helps addressing some of the challenges with the development of distributed reactive systems. We discuss the benefits of service-orientation in all phases of system development from requirements engineering to system operation. We show how service-oriented concepts get applied in various application domains and compare commonalities. We emphasize the critical importance of system architecture to distributed systems, especially for larger size systems and development projects. Similarly, we show how systematic development processes can cut through the thicket of day to day development, by reducing activities to manageable tasks with defined outcome, increasing outcome predictability and quality, and lowering cost of development. We introduce the V-Model XT as a successful system development process model.

In Chapter 3, we describe an existing comprehensive formal system model for distributed system design with services as first class design elements. We make use of the Focus

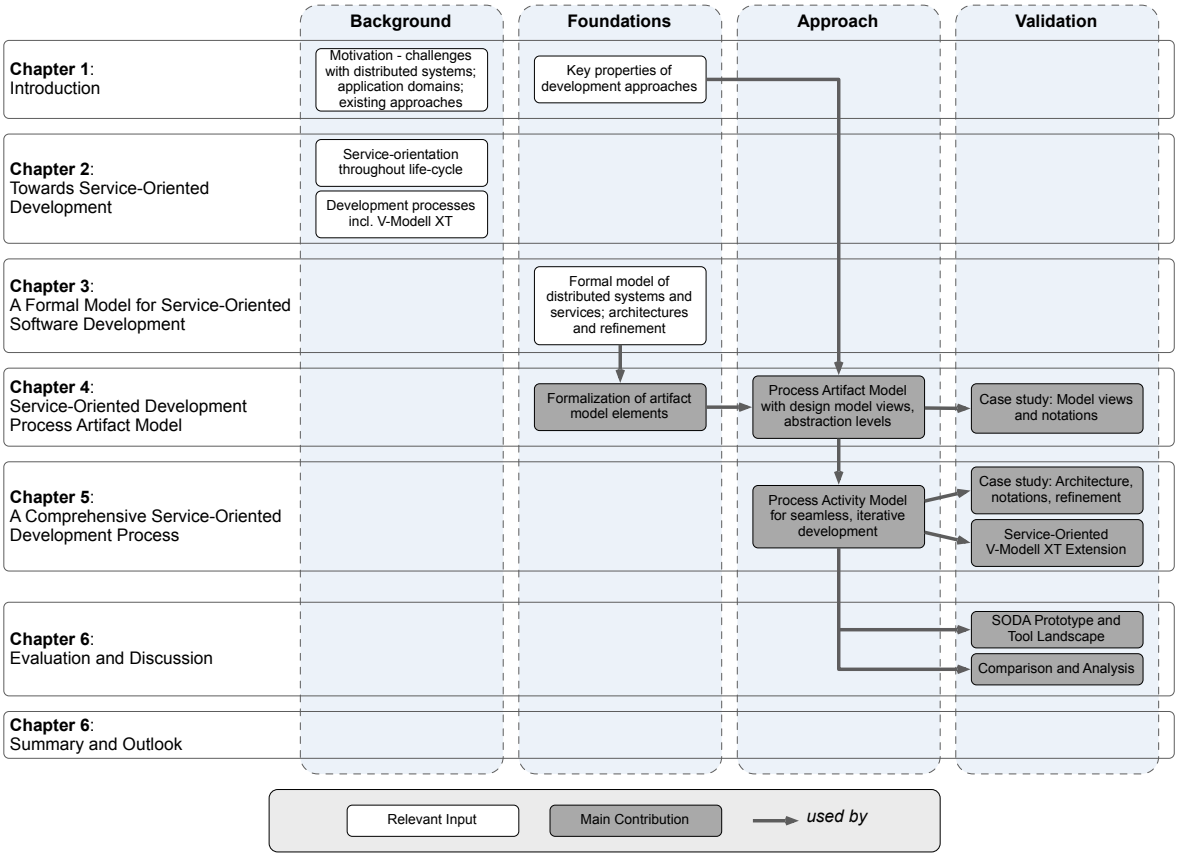

Figure 1.17: Thesis structure showing contributions, background and foundations

formalism and distributed system theory and add formal definitions for hierarchical system architecture, service, and interaction design. We describe a notation based on Message Sequence Charts (MSC) for the specification of services as interaction patterns. We show how hierarchical system architectures can be specified and we provide methodology for the iterative refinement of design specifications.

Chapter 4 introduces the artifact model of our service-oriented development process and one of the main contributions of this thesis, specifically designed to flexibly represent all concepts of our formal model. We show how the notion of service is prevalent on all levels of abstraction in the process, throughout all phases of development. We define process artifacts in the context of the model views they represent and provide comprehensive domain models showing their dependencies. We define concepts and process elements using the introduced formal system model and process metamodel. This associates meaning with syntactic constructs and enables us to reason about distributed systems and the properties they express. We make use of our running example as case example to demonstrate the different concepts, notations and methodology in a comprehensive walk through.

Chapter 5 provides the activity model and completes the introduction of our service-oriented development process. We explain the activities for developing a logical architecture design model, including service and component designs. We discuss methodology for iterative system design, such as model refinement and model refactoring. We then embed our service-oriented development process within the V-Model XT, by providing a modular service-oriented extension of the V-Modell XT as a process component. We show how the V-Modell activities for project management, quality assurance, configuration management and system engineering integrate with and support service-oriented development.

In Chapter 6, we evaluate our process and discuss the findings. First, we present our evaluation strategy that rests on demonstrating automation and practical applicability though tool support, and on comparison with established development approaches. We show practical applicability of our process as a service engineering solution by laying out an architecture for process automation and tool support. We describe the SODA—Service-Oriented Architecture and Design Tool—architecture and prototype that we developed in support of our approach. It manages the entire development process and provides a plug-in platform for specialized development tools. We compare service-orientation to traditional development approaches using the key properties introduced above and indicate where and how service-orientation is an improvement over current model-based development approaches. We also show possible disadvantages and shortcomings of our approach.

We provide a summary for this thesis in Chapter 7 and show an outlook towards future research and refinement of our approach.

1.6 Chronology and Related Work

The development process and service-oriented approach we introduce in this thesis integrates with and extends existing work and was developed over an extended period of time. Roots of our work can be found in several service-oriented and model-based theories. Our goal was to develop an integrated service engineering solution that employs well-known, proven techniques, combining and extending them where needed with suitable theories to provide a comprehensive solution addressing the problem statements for our target domain.

Figure 1.18 depicts important publications in the provenance of this thesis. It also highlights our contributions, indicated by the degree of shading. The figure also lists important developments in the software industry and with Web technologies along the same time axis. We explain influences and relationships to other work in more detail in the sections below.

Significant parts of our approach are based on the work of Broy et al., making use of the Focus [BS01] formalism and its system model as formal foundation. We apply the mathematical model of streams between independent asynchronously communicating systems. Focus is also the formalism for semantics definitions of MSCs [Bro05a], State Transition Diagrams (STDs) [GKRB96] and for service-oriented concepts [Bro05b, BDG+06].

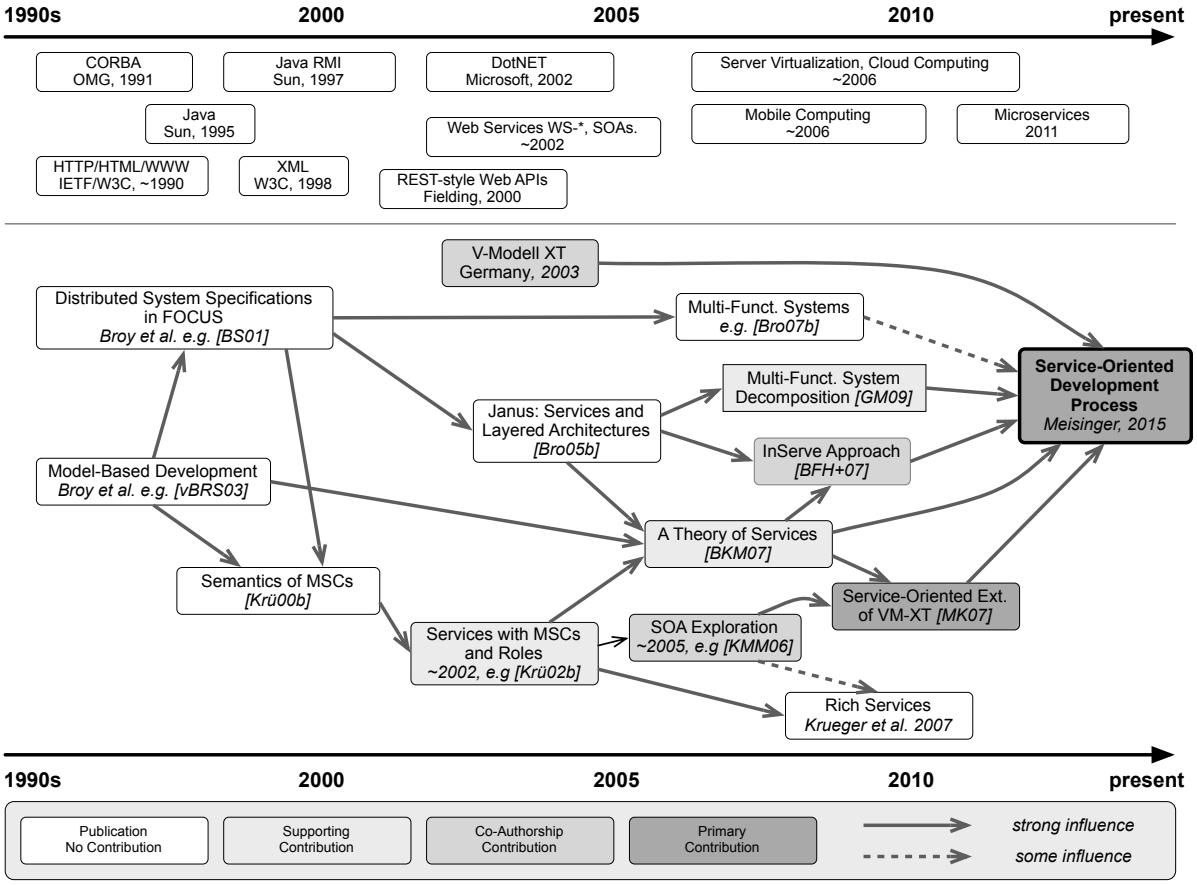

Figure 1.18: Chronology of research, related work and contributions in context of important software industry and Web developments

The support tool AutoFocus [HS97] and similar tools [SFGP05] were built as CASE environment using the Focus method. A broad range of system engineering methods and applications is based on the foundations provided by Focus, such as [HB05, BR07, Bro05c].

We also draw on significant parts of the work of Krüger [Krü00b, KGSB99], a methodology for model-based development of distributed systems using Message Sequence Charts (MSCs) [IT96] as specification technique. This work extends the notation of MSCs and provides precise semantics together with strong methodical underpinning. One key feature is a component synthesis algorithm, which generates component state machines out of a set of separate MSC-based interaction specifications [KGSB99, KMLS05]. Krüger et al. apply this methodology [Krü02b, Krü03, KSTW04, KM04b, Mat04] and MSCs as specification technique in a service-oriented development approach. This approach introduces a role concept that raises the abstraction level [Krü04] of MSC-based interaction specifications by abstracting from concrete deployment components, elevating the specifications to a logical level, mappable to several potential deployment architectures.

We contributed to refining and applying this approach, supported by the DFG sponsored *InServe* project [InS07]. These efforts yielded substantial results for domain-independent service-oriented development [BKM07, DKMR05, KMM05, KMM06, KMM07a, KMM07b] and tool support [AKMP05, KLM06, EHK+07]. Main ideas of the work described in this thesis were developed as part of this project. A distinguishing element of the contribution of this thesis is the combination of development process and hierarchical architecture model, as well as the introduction of component operational modes to structure service-oriented specifications.

Our work relates to applied theories and methods in the context of the FOCUS theory by Broy et al. on the application of services for functional specifications of multifunctional systems [Bro10, Bro07b, GHH07, Deu08, Rit04, Rit08, DGP+04, RFH+05, Bro06, Bro07a] such as applied during the development of product lines [Sch06a], and context-sensitive systems [DGH+05]. In particular the concepts of abstraction levels in system design models influenced our work. We include concepts from the work of Schätz et al., who introduce a formal model of services in [Sch02, SS03, Sch06a, Sch06b, Sch08]. Their concepts of operation modes have influenced our architecture models. Work on software development processes influenced the design of our development process and the integration with the V-Modell XT, such as [GMP+03, Gna05]. Work on requirements engineering [Gei05, HRSW06, Pen11, Vog15] is indirectly related as well. The outlook in Chapter 7 will discuss potential ways of relating the various research results with our work.

This thesis draws upon and aggregates work published by the author on software development processes [GDMR04, MRD+04, MK07, WM06], a formal model of services and functions [BKM07, GM09], service methodology [BDG+06, MR08, KMM07a, KMM08, BFH+07] and tool support [EHK+07, KMMP06]. As member of the V-Modell XT authoring team, we analyzed requirements [DGMR03a] and developed the underlying architecture [DGMR03b, MRD+04] for the process model. We also contributed extensively to its tool support.

During our work on the OOI Cyberinfrastructure [ACF+09, OOI14], we incorporated many service-oriented concepts into the development of this large-scale system architecture and its open source release SciON [Sci15]. We contributed to research related to the monitoring of distributed interactions, cf. [BCD+13] and supported the implementation of a monitoring prototype. This line of research led by Honda, Yoshida et al. is based on a theory of session types and uses a language called "Scribble" to formalize interaction protocols between distributed system components. We also worked on a formalization of domain-relevant relations between distributed components—or agents—such as the negotiation, tracking and enforcement of agreements between agents, cf. [SAB+13] led by Singh et al.

2 Towards Service-Oriented Development

In this chapter, we describe services and service-orientation applied to the development of distributed reactive systems. We show that service-orientation provides advantages throughout system development, in particular when compared to development in more traditional ways. We emphasize the importance of system architecture for distributed systems and introduce service-oriented architectures (SOA). We show how development processes help to manage development complexity and increase the productivity of development activities as well as the quality of the development result; we introduce the V-Modell XT as representative process and outline its extension possibilities.

Contents

2.1 Introducing Service-Orientation

2.1.1 A History Of Services

Software development has changed substantially during the recent five decades. Early computers were programmed in machine code and assembly language, hand-tailored to specialized compute equipment. Subsequently, higher level programming languages emerged abstracting common code patterns. With growing code size, in order to keep uncontrolled code sprawl at bay, Parnas, Dijkstra et al. proposed *structured programming* [Dij70, DDH72, Par79] with an emphasis on control structures and modularity. Dijkstra's letter to the editor "Go To Statement Considered Harmful" [Dij68] became a well known expression of this evolution. The systematic application of structured programming and structured design first enabled the development and maintenance of larger-scale systems.

Object-oriented programming strengthened the concepts of modularity and encapsulation while at the same time introducing ideas such as inheritance and polymorphism. Object-oriented software development methods [Jac92, Boo93, SGW94, GHJV95] brought guidance in form of flexible notations such as class and state diagrams, strategies such as use case driven development [Jac92], and domain specific semantics such as ROOM [SGW94]. The object-oriented movement reached maturity with the introduction of the UML [OMG11b], bringing together competing notations and spawning peripheral efforts such as UML profiles [OMG03b], formal semantics [LCA02] and tools [HIM00, Krü02a, vBRS03].

Component-based development [Szy02, Rau02] addressed some of the weaknesses [Bro03a] of object-oriented development when applied to larger systems: the breach of encapsulation due to subclassing, leading to a reduced ability to define modular reusable units of software development. Software components emphasize reuse and provide clear interfaces specifications. In addition, they are better suited to address parallelism and distribution with each component supporting an independent control flow.

The idea of *services* emerged out of various directions, often refining component-based approaches and concepts. The term "service" mostly refers to a distinct function in a system, brought forward by one or multiple software components. Use cases in object-oriented approaches may be formalized into the services of a system. Services often define interfaces for their use by service consumers. Applying strict interfaces to clear functional responsibility—the service—results in strong modularity; this is one of the convincing arguments for service-oriented approaches. The existence of a service makes no statement about how the service is realized and does not restrict the inner workings of the components participating in providing the service. Services can exist anywhere within a system.

In general, applying any form of service introduces a higher level of abstraction to system design and implementation when compared to more traditional component-based techniques. We speak of *service-oriented development* and *service-oriented architectures* (SOA). The increased abstraction helps to manage the complexity of larger-scale, more

heterogeneous systems. Service-orientation is particularly useful as a system integration strategy, where the challenge is to integrate various incompatible legacy technologies into a coherent, maintainable system of systems. Business processes within information systems, for instance, can be represented and accessed as services, orchestrating an interplay of distributed interacting software components. This interplay can even cross local networks and organizational boundaries.

In the introduction we defined that a service "represents a discoverable well-defined part of functional behavior of an application, coordinating implementation level entities". We use this definition in the remainder of this thesis. In the following, we describe different forms of services in greater detail and show how services contribute to the development of distributed systems.

2.1.2 Categorizing Service-Oriented Approaches

The term service is used often and extensively within academic communities in the software industry. There exist many service-oriented techniques and technologies, often targeting different problems and focusing on different application domains. Few approaches provide a sufficiently precise definition of the term service and of related concepts. Confusion often results when comparing different approaches. Questions such as the following arise:

- What is the problem class the approach tries to solve?
- Which domain is an approach targeted at?
- What are the main concepts of an approach, besides services?
- What are the advantages and disadvantages of the approach?
- Which part of the development cycle is addressed by the approach?
- What is the maturity and practical applicability of the approach?

One goal of this thesis is to provide a more comprehensive view on service-oriented development and its benefits in general. As a first step, we explain a number of specific service-oriented development approaches in more detail.

Service-oriented concepts got incorporated into many approaches and were applied to a number of application domains. In order to compare service-oriented approaches and to evaluate their applicability to a given development scenario, it is important to characterize them according to well-defined criteria. In [MR08], we introduced a list of characteristics of service-oriented development approaches. These characteristics include:

1. Coverage of system development activities
2. Specification of independent functions and functional properties
3. Specification of functional dependencies and interactions
4. Capturing global and local views on services

5. Abstraction from system structure, deployment configurations and technologies
6. Black-box service interfaces
7. Specification of crosscutting properties and behavior
8. Specification of quality properties
9. Specification of forbidden behavior
10. Specification of data flow, transactions and data integrity
11. Partial specifications
12. Support for underspecification (nondeterminism) and refinement
13. Provides methodology or processes
14. Service description and discovery mechanisms
15. Execution infrastructure
16. Support for dynamic execution environments and adaptivity
17. Standardization and community-acceptance
18. Formality and precision
19. Direct applicability

The availability of these characteristics enables a systematic comparison of service-oriented development approaches. Assigning relative weights to these characteristics, suitable for a given development scenario, can help to rank approaches for a more informed selection. Such a contextual prioritization can pinpoint trade-offs where conflicting goals exist in order to meet the project requirements and constraints. For instance, in one development scenario it may be more important to support iterative development (13.) than having full formal support of model refinement (12.). In another scenario having support for partial specifications (11.) may be more important than the ability to specify forbidden scenarios (9.), while the existence of an execution infrastructure (15.) may be irrelevant for the project. For more details, see [MR08].

One of the main distinctions of service-oriented approaches is whether they apply to system design or to system implementation. Figure 2.1 shows a Venn diagram intersecting systems designed and developed using service-oriented methods with service-oriented architectures. We know that many existing service-oriented architectures were developed in traditional ways—many Web Services based systems fall into this category. There are some SOAs, however, for which service-oriented development methods were used. These systems benefited from the full advantages of service-orientation. There exist systems developed in a service-oriented way that don't expose services as structural elements in their system architecture. Examples exist in the embedded systems domain, cf. [AKMP05]. This thesis primarily targets systems developed in a service-oriented way—SOA or not—independent of specific development techniques, processes and application domains.

Figure 2.2 qualitatively illustrates areas of best applicability for service-oriented and other development techniques. The horizontal dimension represents system size and the vertical dimension the number of system functions. The diagram partitions the map into best applicability for object-oriented development, component-based development and service-oriented development. The intensity of the shading indicates better applicability. Pure

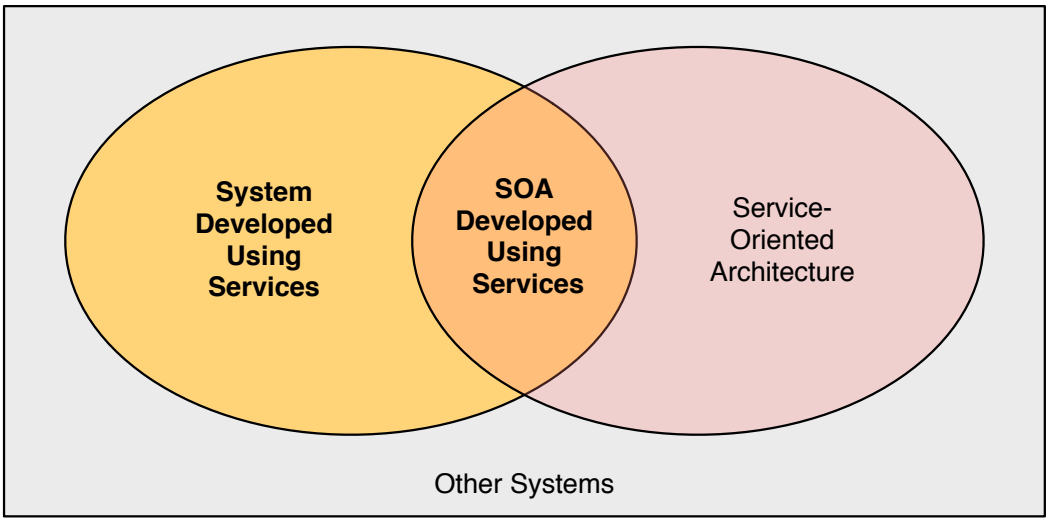

Figure 2.1: Systems developed using services vs. service-oriented architectures

object-oriented development is most suitable for small to medium sized systems. Larger systems and systems with many functions become problematic to manage. Component-based development is most effective for larger systems, in particular for distributed systems, if the number of system functions remains rather limited. For a long time there existed no approach suitable to effectively develop larger, distributed multifunctional systems. Service-oriented development emerged, as described above, addressing the complexity of medium to large sized multifunctional systems. For large scale systems, this technique is particularly effective when combined with component-based methods, e.g. for the deployment of distributed service components.

There are many system architectures that are primarily structured around providing, consuming and brokering services. We speak of service-oriented architectures (SOA). Here, services and service interfaces are first class elements of the implemented system. Such system architectures are often built around a distributed service deployment infrastructure. Web Service based systems [New02] represent the largest segment of SOAs. Message-based SOAs are pervasive as well. Service-oriented architectures are often supported by service infrastructure, such as service registries, service orchestration and choreography frameworks, and Enterprise Service Buses (ESB).

A system architecture can be characterized by its degree of structural and functional decomposition. A higher number of components often implies more distribution: components may realize the distributed system elements that interact via networks. Traditional component architectures structure systems into components, often recursively structured into subcomponents. Service-oriented architectures—providing a number of services to consumers—realize a functional decomposition of a system. Services can make use of other services, as needed to provide their functionality. Figure 2.3 compares both dimensions,

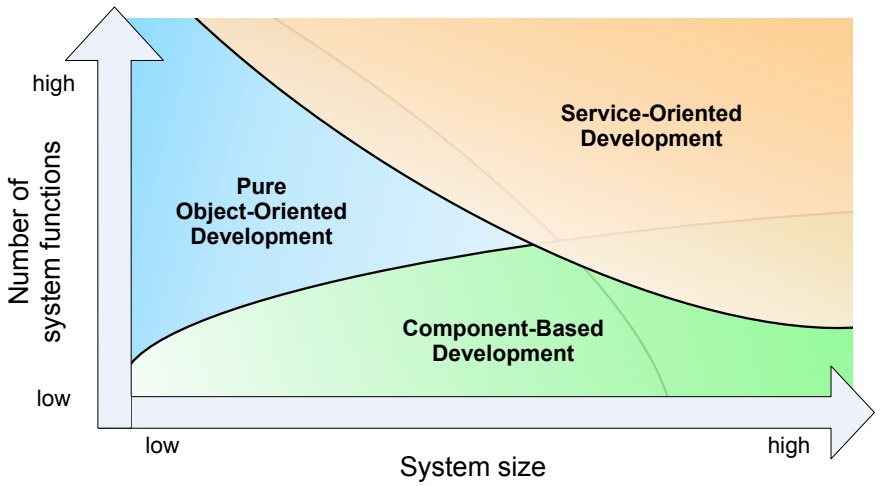

Figure 2.2: Best applicability for object, component, and service-based development techniques

partitioning system architectures into four main categories: monolithic systems (few components and distinguished functions), component-based architectures (many components and few functions), basic service-oriented architectures (many functions with flat architectures) and complex service-oriented architectures (sophisticated distributed multifunctional architectures).

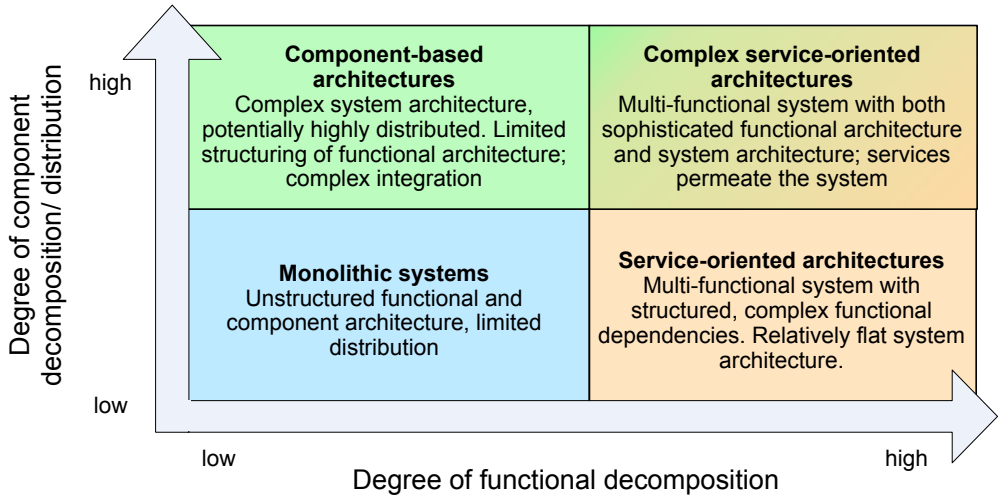

Figure 2.3: Categorization of system architectures by number of components and functions

For highly multifunctional systems with complex target environments and significant operational constraints, such as embedded system architectures with many electronic control units, service-oriented development can help managing the complexities of distributing multiple functions across compute nodes. Nonetheless, the resulting system may be component architectures, not exposing services as implementation entities.

2.1.3 Related Work: What are Services?

Many attempts exist in the literature trying to capture the essence of services and service-orientation. There exists no broad consensus and precise definition but over the years some core definitions and interpretations have emerged that capture the underlying concepts reasonably well. In the following, we briefly summarize related work defining and categorizing service-oriented approaches.

[Hum08] points out that many publications define concepts of service-orientation similarly to established concepts and technologies such as components, interfaces and operations. The author rejects a redefinition of established concepts caused by the "hype" around the topic, and instead sees services as a way of structuring the usage interface of a system into a landscape of business services. These business services can be completely nontechnical, but can be realized by application services provided by system elements. Actual system architectures comprised of components can then be derived from these business and application services. In general, we follow Humm's interpretation. We also see the main contribution of services in structuring a system's usage interfaces systematically, based on a solid technical foundation of component-based systems. In our work we do not distinguish business and applications services and keep concepts domain independent. Rather, we provide a method for iterative specification and refinement of services as system functions into implementable components that can be mapped to deployment configurations.

The work in [AGA+10] has a very concrete technical foundation in the world of business information systems around IBM enterprise architectures and Web Services technologies. It describes a service-oriented development method (SOMA), addressing all phases of software development, leading to concrete Web Services based system architectures that are less brittle to maintain than traditional architectures. The method breaks down system development into seven distinct but interdependent activities, including business modeling, specification and implementation. The method identifies detailed steps for all activities with decision guidelines and references to existing procedures. This work also distinguishes nontechnical business level services and implementation level services, and provides references to core elements of the IBM services architecture. We share the ideas of applying service-oriented concept to the entire system development life-cycle and providing systematic guidance in form of a development process. We keep our approach domain independent and allow a mapping to different target technologies.

Another direction for the harmonization and generalization of service-oriented approaches comes from standardization bodies such as OASIS. Here, interoperability between service-

oriented systems is the highest goal. The Web Services Architecture [W3C04b] provides a technical recommendation for designing system architectures based on common concepts and infrastructure. We will discuss service-oriented architectures (SOA) in more detail in Section 2.3.1. For information systems, the emergence of Web Services influenced many system architectures. Service-oriented architectures promised to provide a viable solution to many problems of interoperability and managing loosely coupled distributed systems [Bar06]. Large-scale systems of systems integration common for information systems shifted towards service-oriented techniques. Such systems are inherently heterogeneous, spread across different domains of authority and resist any form of tighter coupling. Legacy mainframe systems need to be integrated with multi-tier information systems, Internet enabled Web front-end applications using enterprise integration middleware. Encapsulating entire systems through services that communicate asynchronously on reliable communication channels is an effective integration strategy [HW03, TRH+04, OHE97, SUN06].

A significantly different point of view on services was brought forward a decade earlier in the context of large-scale telecommunications systems. As described in the introduction, these systems provided the intelligent communication networks for large communication infrastructures and telephony systems. Telecommunications systems were structured generally in form of a capable routing and switching infrastructure, providing a variable number of "intelligent services" or features [Fre04, Zav93] to end users. Features were specified by the inputs they consumed and the features they depended on. Such systems often implied an inherent dependency between their features; architectures were geared towards avoiding unwanted feature interaction, often caused by overlapping interpretation of user input signals. Feature dependencies and feature interactions can be managed on the level of feature design, for instance by applying a prioritizing strategy [GM99, DH99, Tur98, JZ98]. Network communication protocols use the notion of service as abstraction of physical and virtual connections between systems; for instance services can serve as access points for functionality between the different layers of protocol stacks, cf. [HB05].

In embedded systems, for instance automotive control systems, software complexity increases dramatically caused by the increase in the number of distributed electronic control units (ECU) and the number of functions that require a collaborative interaction of these ECUs. New innovations in automotive systems are largely created by integrating previously independent functionalities. This leads to vastly complex, unverifiable systems that threaten to obstruct further system growth and quality. The industry moves towards system architectures that isolating the functions of such systems from their actual execution environment. AUTOSAR [Aut06b] is an example of such an architecture inspired by service-oriented ideas. Service-oriented ideas are also prevalent in automotive infotainment systems.

Broy et al. provide a general theory of services with a formal interpretation of service-oriented development and service-oriented architectures [BKM07]. They define the concepts of service, service interface, service combination etc. using a stream-based formalism and a formal model of distributed component architectures. Our work is building on this

foundation and places service-oriented development into an engineering context with a development process, while remaining domain independent. See Chapter 3.2 for a formal service definition based on this theory.

2.2 Services During System Development Activities

In this section, we describe how service-oriented methods can be applied with advantages during system development activities. These activities include:

- *Requirements Engineering,* defining precisely what is expected from the system, documenting existing constraints, while developing an understanding of the problem space, current situation, expected system purpose, and user and stakeholder needs. This activity results in a structured, prioritized list of testable, consistent requirements that can be managed as stakeholder needs change, and that can be traced to design and implementation artifacts. A "formal system specification" represents a very precise form of requirements.
- *System Design,* developing the structural and behavioral blueprints of the system. This activity includes defining candidate system architectures and designs that meet the system requirements, assessing feasibility and effectiveness of these candidates, breaking down problems into manageable units of implementation, defining system interfaces, and providing maintainable system design documentation.
- *System Implementation,* translating system designs into executable software. This activity includes refining designs into implementable fine-designs, implementing and delivering software modules and components, and developing unit tests for these.
- *Integration and Verification,* combining implemented software modules and components into larger components, subsystems and the entire system according to an integration plan. This also means making sure the built system meets requirements and designs.
- *Deployment,* establishing an integrated system in a target execution environment and preparing for continued operation of the system. This activity includes installation, configuration and initialization of the system.

The order of these activities exposes a natural progression from problem definition to solution. There is, however, no fixed ordering in which these development activities must be carried out. For smaller, well defined development efforts, the activities can almost occur in sequence with minimal overlap. For larger projects, more incremental or iterative strategies help to keep development complexity manageable and adaptable to changing stakeholder needs and technologies. Here, many activities occur concurrently, but may be emphasized or toned down according the current development phase.

Development processes help to increase repeatability of system development. In Section 2.4, we introduce important exemplars of such processes, including the V-Modell XT, the Unified Process [Kru00a, JBR99], and the ISO 15288 standard.

Services play an important role during all development activities, when developing in a service-oriented way. Figure 2.4 summarizes important purposes that services fulfill over the course of system development.

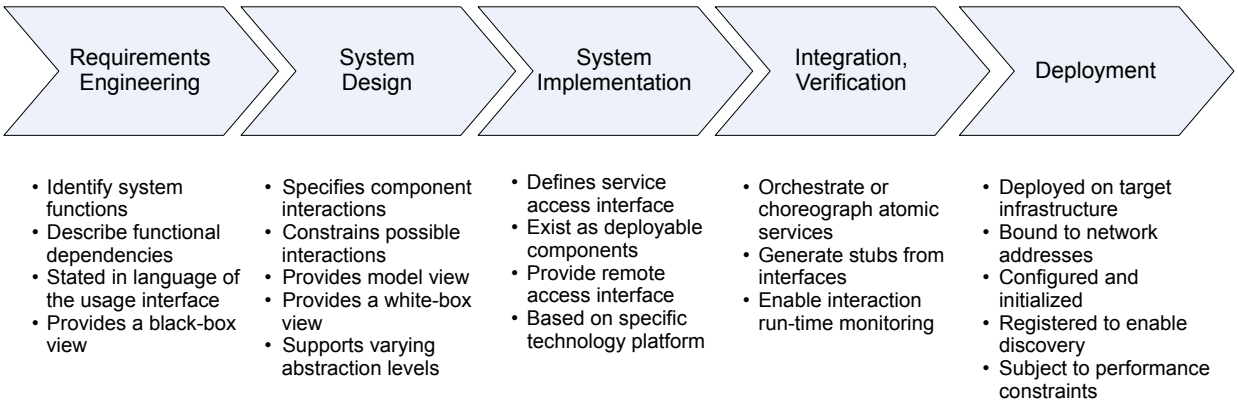

Figure 2.4: Service purposes during main development activities

In the following, we briefly characterize the main development activities and describe the benefits that service-orientation brings, cf. [BDG+06].

2.2.1 Services in Requirements Engineering

Requirements engineering [SK98] (RE) targets the systematic elicitation and management of requirements for systems under development. This activity is concerned with capturing the functional and nonfunctional properties of systems in a structured way. Typically, requirements are captured and analyzed in natural language, using software tools such as word processors, spreadsheets, IBM DOORS[1] etc. Scenarios of system use can be captured in form of use cases. Relevant activities cover a wide variety of tasks that create artifacts ranging from informal to strictly formal, typically using the language of the problem domain. The goal of requirements engineering is to provide a viable basis for subsequent development efforts: ideally a requirements specification that is complete, consistent, concise, understandable and flexible to change. Significant work has been published on methods and formats for capturing such requirements specifications, cf. [Som89, Wie99, Bro97, Gei05, SFGP05, Pen11]. Other requirements engineering activities in the early phases of system development include domain analysis [Eva03] and requirements structuring [vL04, DvLF93].

[1] `http://www.ibm.com/software/awdtools/doors/`

Raising the level of abstraction as means for managing more complex systems has influenced requirements engineering. Service-oriented concepts can be used to capture and structure the usage behavior of systems as first-class concern [HRSW06, Deu08, Rit08]. Related approaches capture the functional dependencies of a system independent of its actual implementation architecture, for instance in form of function networks or as a collection of features [Blo97, ZJ97]. Use case and scenario-driven approaches [Jac92, WK04] are similar, but tend to be less formal. Goal-driven requirements engineering approaches, such as [DvLF93, vL04], try to establish sophisticated models of a system's problem domain as specification. They are based on a specifically designed goal-driven metamodel. Goals are the intents and objectives of a system that can be achieved when pursued by actors, any active entities, and when coordinated using events.

Distributed Feature Composition (DFC) [JZ98, ZJ03], for instance, is a feature-oriented virtual architecture for the specification of telecommunication services. With perfect behavioral modularity it would be possible to arbitrarily change the behavior of a system by composing features. However, there exists the problem of feature interaction, as "some way in which a feature or features modify or influence another feature in describing the system's behavior set" [Zav01]. The authors distinguish between wanted and unwanted feature interaction. Unwanted feature interaction may result in inconsistent, nondeterministic and not implementable specifications. The feature-oriented methodology of [Zav01] applies formalization to the specification of features, their dependencies and the underlying architecture. This enables analysis of feature specifications for feature interaction and a precise behavioral specification of feature-oriented systems. The DFC architecture has been formalized in a combination of Promela and Z. Specifications in Promela can be verified using the SPIN model-checker [Hol03] for safety and liveness properties.

When performing requirements engineering using service-oriented concepts, services represent the functions or features of a system provided to the environment, for instance functions required to satisfy a business process describing an overarching system use case. As [Eva03] points out, it is important to acquire an understanding of the domain the system is targeted at. Similarly important is an understanding of the basic functions the system provides to the domain. Evans proposes the use of services on a high level of abstraction in system and software architecture design and modeling. Multifunctional systems offer a number of independent services to the environment, which often makes them complex to describe and decompose in a straightforward way. Each service provides a partial view on the system behavior. The entirety of services makes up the full system functionality.

Services can depend on other services. Services may, for instance, "interact" or "control" or "influence" other services. These dependencies can be depicted in a service dependency graph, cf. [Rit04, Rit08, Vog15], and analyzed for consistency and certain properties prior to system design. Certain forms of feature interaction can be detected by analyzing a service dependency graph. The more precise a service specification, the better it is suited as baseline for system verification and for system design refinement. A detailed, precise service specification can describe the full usage behavior of a system. Design specifications

and implementations can then be traced and later verified against this service specification, for instance to show correctness and completeness.

For a certain class of systems or system components, such as for safety or security critical systems, it is required or desirable to define a *formal system specification*. Such formal specifications are expressed using formal languages and formalisms suited to capture wanted or undesirable properties of a system. Formal verification of design specifications or implemented system elements can then be performed against the formal specification. Creating such a specification typically requires a substantial investment of time and project resources and is often accompanied by a rigid development process. In particular, all system functions and nonfunctional properties need to be known before larger parts of system design and implementation can commence. A given formal specification can be analyzed for consistency; certain properties of the system can be proven present or absent, providing the underlying formalism is strong enough. A precise service specification mappable to such a formalism can provide the basis for a formal system specification. Additional nonfunctional properties may be associated with the system or any of its services.

Product line engineering [Sof07, PR02, Sch06a, Gru10] is a way of reducing cost and time to market for software systems with a common set of features and core assets [Sof07]. Product line engineering becomes more effective the greater the overlap in shared functions and the greater the degree of function reuse. If a system can be specified as a set of partially related functions, a product line variant can be defined developing a modified configuration reusing some of the function definitions. Ideally, all different configurations are based on the same repository of functions or features; so called variation points clearly define where actual differences are possible. Not all potential combinations of features are valid. The feature model therefore must also include feature dependency information, for instance in the form "Feature A implies Feature B and not Feature C". A precise service specification can provide an effective basis for a software product line, cf. our work in [KMM05]. Here, system functionality is decoupled from system architecture and implementation technology choices. System functions (services) can be combined into product line variants, which themselves can be deployed on different target architectures, by providing a different deployment mapping. This work can be leveraged to evaluate different architecture candidates of product line variants, by generating and comparing prototypical implementations [KMM05].

We see our service-oriented development approach related to formal requirements engineering. Our approach combines the specification activities of service-oriented requirements engineering with the ability for a seamless refinement of formalized requirements into design models and then fine-designs. One of the distinguishing characteristics of our service notion is a formalization based on a generalization of distributed components as partial behavior functions. Services remain independent and can be combined freely while not yet mapped onto a component architecture. Subsequently, the services model changes from a *partial* to a *total* behavior specification, defining system reactions for all inputs and error conditions. Structural and behavioral details can be refined, for instance to reduce nonde-

terminism and to a split a component into subcomponents. The benefit of our approach is that there are no changes in modeling technique and concepts between requirements engineering and system design. We will elaborate this in Chapters 4 and 5. The outlook in Chapter 7 will mention additional possibilities for combining formal requirements techniques with our approach.

2.2.2 Services in System Design

System design activities target developing comprehensive, consistent and detailed structural and behavioral blueprints of the system-to-be, ideally unambiguous and readily implementable. This includes developing candidate system architectures, assessing feasibility and effectiveness of these candidates, decomposing designs into manageable units of implementation, and providing maintainable system design documentation. System design artifacts ought to be traceable to requirements and later to implementation artifacts.

Services can play a substantial role during system design. Designing a system in a service-oriented way provides designers with a means to abstract from implementation details in early phases of system design and focus on the functions provided to the environment and their interdependencies. We will show this in the following.

System Architecture A definition of the essential structural and behavioral elements of a system determines a *system's architecture*. A system's structural elements may include software and hardware units. For software-intensive systems, it is their *software architecture* that dominates the system architecture. [CBB+10] defines: "the term software architecture denotes the high-level structure of a software system. It can be defined as the set of structures needed to reason about the software system, which comprise software elements, the relations between them, and the properties of both elements and relations".

Designing a sound system architecture is an important step in system design, with direct ancestry in the early concepts of Dijkstra's "structured programming" [Dij70], and Parnas' "modules [Par79]. Interfaces, both behavioral and informational, provide an important means of imposing system architecture on a system. System architecture becomes even more important as systems increase in size, complexity, distribution and heterogeneity. The choice of system architecture most always influences the basic nonfunctional properties of a system, such as performance, scalability, reliability, security, robustness, maintainability etc. Certain architectures turn out to be more effective in supporting the capabilities of a system than others. It is the expertise of the system architect combined with a suitable system design process that is critical to making the right design choice. International standard IEEE-1471 defines content requirements for architectural descriptions of software-intensive systems, including the presence of modular, first-class, consistent views, addressing the concerns of all system stakeholders [Arc00].

System architectures developed using a service-oriented design approach promise to be, in general, more flexible and robust to changing environments and requirements. Systems are designed as composition and orchestration of services of different granularity, structured in form of a service architecture. A supporting system (component) architecture structures the underlying implementation and operational environment for the services. Such architectures are less tied to specific implementation choices and technical details.

In traditional system architectures, both component deployment and network layout are static and do not change while a system is in operation. Deployment and operations activities become more complex, however, if the system supports dynamic network and deployment layouts. "Cloud computing" [KTMF09], for instance, enabled by virtualization technologies, offers systems a virtually unlimited pool of compute resources on demand, thereby enabling flexible adjustments of systems to load. We speak of "elastic computing" [MKF10, CAF+09] when systems regulate their footprint based on system load to meet user need. When new system components get deployed on new virtual compute nodes, they "contextualize" and join the system, increasing available capacity and modifying the communications links within the system. System architectures that support elastic computing are more complex to design and verify, due to the high degree of interaction variability. A different form of dynamic system architecture exists with mobile computing. Here, execution environments, network links and locations of components—sometimes called agents—are not predetermined and change over time. Contextual information about a component's environment and infrastructure may influence the behavior of the application, increasing the complexities of designing and verifying such architectures. Several "integration patterns" [HW03] emerged to manage interaction complexity, such as publish-subscribe information flows using shared work queues. In addition, service-oriented design and implementation approaches can help managing system complexity. Dynamic and adaptable architectures are often service-oriented because they also strongly benefit from developing independently deployable, precisely specified units of behavior, cf. [Sal02, Fah05, Bro14]. The combination of both strategies offers powerful tools to system designers.

In this thesis, we focus on static system architectures for fixed target deployment environments. We do not model and consider architectures that dynamically add and remove system components as described above.

Design Models and Design Documentation When describing system designs, it is important to provide suitable views [CBB+10] on the system as described in the introduction. Model-based development [SPHP02, BvRS02, Bro05c] provides these views and helps managing the complexity that comes with the development of distributed software systems, as described in the introduction. Models provide powerful abstractions during system development for understanding the complex interdependencies between system elements, especially during system design. Models can provide multiple consistent views on the system and its architecture on different levels of abstraction, tailored to specific needs of designers during system development.

Services can be used as a way of structuring the modeling process and the design models. In such an approach, services are the design entities that drive architecture development and component identification. Services capture defined pieces of functionality and can be associated with elements of the software architecture, e.g. components, patterns. Often, services can be seen as functionality provided by an interplay of collaborating architectural entities. An effective way to describe interactions between independent entities are sequence diagrams or MSCs [KMM06].

When working with design specifications and design models it is important to choose suitable description techniques and modeling languages. The notation determines whether working with designs is intuitive, maintainable, allows an iterative approach to system design and provides a good basis for subsequent system implementation, integration and verification. Many notations exist from academic and practitioner backgrounds; many of them come with associated methodology, ranging from limited to dominating.

Model-Driven Development (MDD) techniques and tools help to specify, analyze, optimize and verify distributed real-time and embedded systems. [SBK+05] show that MDD techniques can significantly improve quality and productivity for such systems, in particular in deployment scenarios for component-based middleware platforms where Quality of Service is important. The authors introduce a platform independent component modeling language (PICML). AIRES [KWGS03] and VEST [SZP+03] are MDD analysis tools that evaluate whether certain timing, memory, power, and cost constraints of real-time and embedded applications are satisfied. The tools enable the annotation of components from a predefined library with real-time properties and a subsequent mapping to hardware platforms where the analysis is performed.

The *Unified Modeling Language (UML)* [OMG11b] is the most well known representative, consisting of a set of notations for the specification and design of software based systems. The UML languages address different purposes and are unified by a common metamodel and action-based semantics [OMG11b]. Most of these languages have graphical notations as primary form of representation. The UML also provides textual representations for all languages based on OMG's XML Metadata Interchange (XMI) [OMG11a], enabling interchangeability between UML supporting tools. Important UML languages and their graphical notations include:

- *Use Case Diagrams:* Mostly used during requirements analysis, use case diagrams express dependencies between the core use cases of the system and the participating actors. Use cases themselves are expressed in a structured textual form. Some development processes make use cases the primary form of requirements specification.
- *Class Diagrams:* Depict the structure of a system by showing classes of system entities and their dependencies. Entities may represent object classes in an object-oriented implementation or abstract concepts in a domain model. Classes can have attributes and operations. Dependencies may occur as association, aggregation, composition, realization and generalization. Class diagrams are the most widely used language of the UML.

- *Collaborations:* Visualize the behavior of a system consisting of multiple interacting entities in form of sequence and collaboration diagrams. These diagrams show dependencies between entities as well as sequences of exchanged information messages.
- *State Diagrams:* Describe the changes of behavior of system elements in form of state machines, consisting of operational states and transitions between these states triggered by observed events.

One of the criticisms of the UML is that its languages overlap in purpose and provide a too rich syntax. It does not define well enough how notations delineate and how they interface when used jointly. Earlier versions of the UML were lacking a semantics for the interpretation of the meaning of the notations. While this was corrected in version 2 of the specification, the general lack of guidance and methodology in the application of the various languages remains. This provides great flexibility, supports broader applicability and enables more intuitive use. Such a lack of clarity and interpretation, however, is undesired for the precise specification of distributed systems; it leads to ambiguous specifications and the introduction of inconsistencies during system development.

A variant of UML focusing on the specification of reactive embedded systems is RSDS (Reactive System Design Support) [LCA02], applying state machines and class diagrams with invariants, backed by a temporal logic based formal semantics. One particular strength of this approach besides the UML compliance is that it is self-contained and does not require the use of any other notations and formalisms. Invariants in RSDS resemble operational modes, as for instance known from continuous control systems [GK82]. Invariants are distinguished into system and environment constraints and into critical and noncritical. Methodology exists, such as refinement operations for specification decomposition.

Other prominent specification and modeling languages include Message Sequence Charts (MSC) [IT96], SDL [EHS98] and SysML [Sys06]. Some were developed for specific application domains, such as ROOM [SGW94] for the modeling of real-time embedded systems. Academic work exists describing system architectures, cf. Wright [AG96] and Rapide [LKA+95]. The focus of these languages is to formally describe components of a system and their connectors, in order to analyze and compare system architectures.

We make use of several notations in our approach. Later in this thesis, we will introduce and apply a specific MSC dialect for the service-oriented development of distributed reactive systems.

Hierarchical Refinement and Composition In particular for larger-scale development efforts, it is important to apply systematic development strategies such as hierarchical decomposition, component modularity and reuse. This helps to increase the effectiveness of development and the quality of the result. Two operations applicable to system design specifications are key to systematic development: composition and refinement.

Composition is an operation that yields a more capable system component out of a number of existing smaller components. If a design approach is "compositional", then a system

composed of well-described components exhibits the properties that are resulting from a composition of the components, cf. [Jon99]. Compositionality relates to modularity in system engineering" [Bro98]. Breaking a system or a component into multiple pieces is called *decomposition*. Both operations are valuable in a typical development project. In many cases, more capable or more targeted components can be assembled out of a set of existing, potentially off-the-shelf components. In other cases, existing specifications are too large to be manageable and they can be broken up by systematic decomposition.

Refinement is an operation to increase the level of detail in a design specification, while respecting certain classes of properties. For instance, [Bro98] introduces several notions of refinement that each target a different purpose. Property refinement enables an addition of capabilities (i.e. new requirements) to a given specification. Glass box refinement enables the addition of implementation detail to a specification, while keeping the externally observable interface. Interface refinement enables the restructuring of a specification, such as the modification of communication channels, while keeping the behavioral properties of the specification constant. This enables to relate design models on different abstraction levels. In a compositional design, "a refinement step for a composed system is obtained by refinement steps for its components".

A requirement for the effective application of refinement and composition is the existence of a sufficiently precise system design model and underlying formalism and theory. The more expressive the formalism, the more classes of properties may be proven respecting refinement and compositionality. In practice, however, a more expressive formalism often requires more rigid development processes and notations that may be difficult or costly to follow. A systematic development process supported by tool automation can help to push the boundaries here, as we will explain below.

Compositionality has many practical applications during system development. When reasoning about a system's properties, it is possible to reduce global reasoning to local reasoning about components [Bro98]. Properties that can be proven for individual system components will hold true on a composite, which can be the system itself. The enables the specification of independent modular system components and directly supports component reuse. Additionally, component specifications can be refined to yield more directly implementable operational specifications, such as deterministic finite state machines. A composed set of correctly refined components will satisfy the properties of the original specification. This is similar for implemented system components. These can be verified for correctness against their component specifications. The integrated system composed out of the components will respect the properties of the design model. Implementation of the individual components can be performed independently. Developers need no knowledge other than the specification of their component—mostly its external interfaces—to complete their task and to verify the correctness of the implementation. The integration of verified components, or a replacement of individual components during the lifetime of a system, becomes substantially easier.

Figure 2.5 illustrates the basic concepts: component specifications (i.e. design models) S_1

through S_4 exist. Properties can be proven for each of these components. The components support interaction with the environment and with one another via compatible communication channels. A system design specification S can be created by composing S_1 through S_4 using a defined composition operator. Additionally, each specified component S_i can be refined (i.e. implemented) into a realized component R_i. This refinement can be verified, see Integration and Verification activities, below. After integration, a realized system R results. The question is, how are S and R related? Given a compositional system design approach, R implements the properties that were true in S.

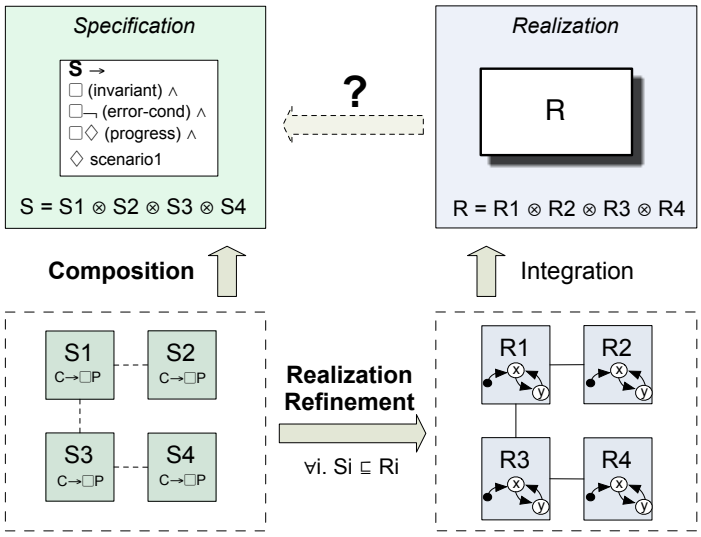

Figure 2.5: Composition and refinement

What does composition mean for services? In [BKM07], the formal theory underlying our service-oriented approach, services are initially modeled as partial specifications of behavior of a set of components. Partiality here means that any of the components may interleave any other behavior, as long as the service behavior eventually remains satisfied. To create more capable services out of existing services, an operation exists to "combine" partial service specifications similar to composition for a set of components. Service combination has slightly different formal properties than component composition. A partial specification can be "totalized" for any given component, i.e. the set of outputs for all input sequences of the component comprehensively specified.

In other service-oriented approaches, services represent behavioral interfaces of distributed components. Composition and refinement as described above can be applied to these system models. Service orchestration in distributed communication environments often requires a more complex approach, because of the need to introduce a separate "orchestration coordinator" component. Given that, individually accessible services and orchestrator can be composed, assuming a reliable communication infrastructure.

In our process, we apply the notions of hierarchical decomposition of the system into

components, and the composition of components out of existing components. Similarly, we support several notions of refinement in addition to service combination. In Chapter 3, we will introduce the formal underpinnings that enable these operations.

2.2.3 Services in System Implementation

During system implementation, programmers create implementation artifacts, such as components, classes, services and modules, following more or less rigid design specifications. This step requires the programmer to translate structural and behavioral blueprints into syntactically and semantically correct executable representations. During this activity, the developer adds substantial technical depth and compensates for gaps in the design. Most commonly for software systems, this means developing program source code in a programming language, codifying the specified component behavior. More often than not, system implementation requires the use of existing code libraries and middleware frameworks. In case of model-based development with comprehensive and precise design models, code generation may be performed (semi-)automatically. For hardware development, the overall implementation goals are similar but the steps to get there are different.

The developer may interpret underspecified designs as long as the specification remains satisfied by the implementation. Successful completion of the implementation of a software component typically requires that the source code compiles or executes, a test against the specification exists—more or less strictly enforced—and the program code be submitted into an integration or automated build environment. Code version control applies in most cases directly as part of submitting source code into the integration environment. Implemented system components can be verified against a precise design specification.

Given a partial design specification, it is possible to apply a "totalization" operation to derive a fully specified component design. A suitable design specification notation and a capable underlying formalism are required for totalization to occur. This enables the system designer to leave a system design open, extensible through additional partial design specifications, and transitioning to a complete behavioral specification at the time of implementation. Totalization, for instance as applied in our approach, entails the removal of nondeterminism and underspecification and making the component input enabled by providing catch states and error conditions for unexpected inputs.

System and component implementation can occur without explicitly specified designs and requirements, for instance during early prototyping or in agile development processes. Here, the joint knowledge of the development team, acting both as design and as implementing team, compensates for a documented system design. It is quite important in such cases that the system remains always executable and gets modified only incrementally in order to provide more immediate feedback of successful implementation and feature integration. Also, a customer or user representative available to the development team can provide immediate feature validation and feedback. Implementation without explicit design can

be very effective, especially when leveraged on a component level, for instance embedded within a larger scale more systematic development effort. Agile approaches often fall short for larger distributed or embedded systems, yielding unreliable implementations. We will discuss different development process strategies below; in Chapter 5 we will suggest a service-oriented process that includes beneficial properties from both design-driven and agile worlds.

Services can play a large role during system implementation. They can be the units of implementation, providing usable interfaces to the environment. As mentioned above, it is possible to implement services without modeling the functions and function dependencies explicitly, but it is preferable to have a service-oriented design model for most seamless development. It is also possible to implement a component architecture based on a service-oriented design model without exposing services as structural system elements.

Multiple technologies exist enabling the implementation of services. Web Services [STK02], for instance, are based on common Web conventions, standards, protocols and tools that enable service interoperability. Independent functions of a system can be exposed as Web Services, accessible via the Internet, for Wide-Area Network integration of interoperable systems and within a system. One of the success factors of Web Services is that services can be provided by anyone, anywhere as long as the required service interfaces and necessary quality of service guarantees, such as availability, are met. Web Service interfaces are defined in WSDL [W3C01]. Development environments and runtime tools such as DotNET [PB01] and Java's Glassfish [Ora13] strongly support creating service-oriented distributed applications. For instance, these environments help to expose functional interfaces of existing components or objects as remotely accessible Web Services; they will generate all interface definitions, stubs and remote clients to make this step as seamless as possible.

Once a service is exposed, the implementation behind the service interface remains encapsulated. Such service implementations typically do not depend on caller or environment state and only process the information provided with the service invocation. Service-oriented middleware can provide local or remote access to the services and systems or systems of systems can be structured as an interplay of multiple service invocations, also known as a service orchestration. Providing services as high-level, remote accessible, stateless encapsulations of functionality supports the decoupling of distributed system entities. Interactions between system elements become explicit and manageable at a level that enables system interoperability in heterogeneous system environments. The use of explicit, widely accepted standards, such as XML [Sha02] and SOAP further help to decrease the burden of system integration. Messaging infrastructures providing service access endpoints further decouple system elements by making interactions asynchronous, i.e. not requiring the immediate availability of all interaction participants. Besides enabling interoperability, another major benefit of service implementations is that services can be modified and even replaced as long as they remain true to the service interface. This makes systems much easier to maintain and manage.

Our approach will support a seamless transition from a service-oriented design model to a service-based or component-based implementation. Our process provides traceability between design and implementation artifacts.

2.2.4 Services in Integration and Verification

System integration comprises the activities of assembling off-the-shelf and developed software and hardware components into coherent larger components or assemblies. The final integration step results in the full system. The existence of an integration plan is beneficial for the manageability of the integration activities. In the absence of a rigorous design and interface control process, interface incompatibilities—syntactically and behaviorally—may be detected only during integration. This may require a modification of one or multiple components and a subsequent retesting. Error resolution at this stage in development is rather costly and can be avoided by placing more emphasis upfront during design activities. Integrated system elements themselves need to be tested for satisfaction of design specifications and requirements.

The integration of standalone, potentially heterogeneous systems, or of existing components is partly a system implementation problem as described above. In some case, a third new component needs to be designed and implemented, functioning as the binding layer between two existing components or systems. Alternatively, an integration middleware layer, such as an Enterprise Service Bus, may take on this binding role as we will explain below, reducing the need and cost for developing and maintaining the binding layer.

Verification is a quality control activity that complements integration, targeting the question "was the system built right?". The system is measured against the requirements and design specifications. Verification and test procedures may exist, derived from specifications, documenting steps and expected outcomes for proving satisfaction of the specification. Verification can be a manual process or can be automated given rigorous specifications and an existing system test harness. Significant work has been done in the area of verification. Verification covers many different strategies and techniques, such as model reviewing, model-checking, theorem proving, testing and runtime verifying implementations, just to name the more prominent.

Validation, in contrast to verification, targets the question "was the right system built?". It typically involves system stakeholders and is a more informal activity contrasting system requirements and intended use case scenarios against the integrated system. The presence of a "customer on site", such as recommended by many agile methods, avoids delaying validation of the system for too long, reducing the impact of change in case the customer is not satisfied.

Services can play a strong role in system integration and verification and can help making validation easier. In fact, integration is one of the strengths of service-oriented architectures. Services encapsulate self-contained units of functionality, accessible only through

a specified service interface. Exposing and consuming services through their interfaces is a task well supported by service-oriented architectures and supporting middleware. The rigor of exposing functionality as services—once done—avoids invisible cross-dependencies between components, facilitating initial integration and long term maintenance of components.

If a service-oriented design exists, with services representing formalized use cases, then integrated systems can be verified against that design. The more formally specified the services, the more automated this verification process can get. Feasible methods are, for instance, model-checking and runtime monitoring. In runtime monitoring, for instance, an existing system can be augmented with a runtime monitor that has knowledge of all system service specifications. All service invocations are monitored for compliance during system operation. Any violations are either reported or prevented, dependent on the intended use.

As a precise model-based system specification and development approach, our work is related to verification and quality assurance. Verification can occur in many forms, with different strategies and techniques, cf. [Sch04, Pel01]. In most cases, however, a precise system specification is the basis for verifying the model and any implementation following the model. As we model logical system architectures and interactions among distributed components, we can check implementations of this model against the specification, for instance by runtime verification [KMM07b]. We can also verify properties of relevance directly within the model, for instance by model-checking [CES83]. In this thesis, however, we concentrate on the engineering tasks and approaches of designing quality constructively into systems; rather than analytically assessing quality of implementations and models.

2.2.5 Deploying Services to Operations

Deployment is the activity of preparing the integrated system for operations in its target execution environment. Steps during this activity include configuring the system for the specific environment, installing the components of the system on compute nodes, installing dependencies, setting up monitoring applications and configuring networks and firewalls. In case a system succeeds a prior system, data migration may be necessary. The system typically needs to complete final acceptance tests before going into operations.

Operations is the activity of maintaining the nominal system function towards users and other systems. This requires operators to monitor user access, system load, resource consumption, the underlying hardware and any system failures. Failed hardware components need to be replaced and defects need to be reported. In many cases, one of the central goals of operating a system is to increase system availability by limiting unexpected system downtime. Tuning the system and its resource use, e.g. by adding database indexes can help increasing system responsiveness and performance. In the presence of defects, maintenance fixes for the system may need to be installed and tested.

Services offer strong benefits during the operations phase of a distributed system. Services as part of service-oriented architectures realize distinct, modular, deployed system elements, cf. Figure 1.14. As such, services can be deployed, initialized, registered, discovered, brokered, and accessed individually. Services can also be replaced without affecting the integrity or availability of the system.

A central feature of service-oriented architectures is the ability to dynamically discover, i.e. look up and find, services during runtime through a service registry. Service providers can register their services with the registry. Conversely, service consumers can find registered services and then bind to them for subsequent use. The consumer can choose available services based on the published service capabilities, available metadata annotations and quality of service guarantees. Figure 2.6 depicts this relationship, often called the "service triangle". Section 2.3.1 below provides more details.

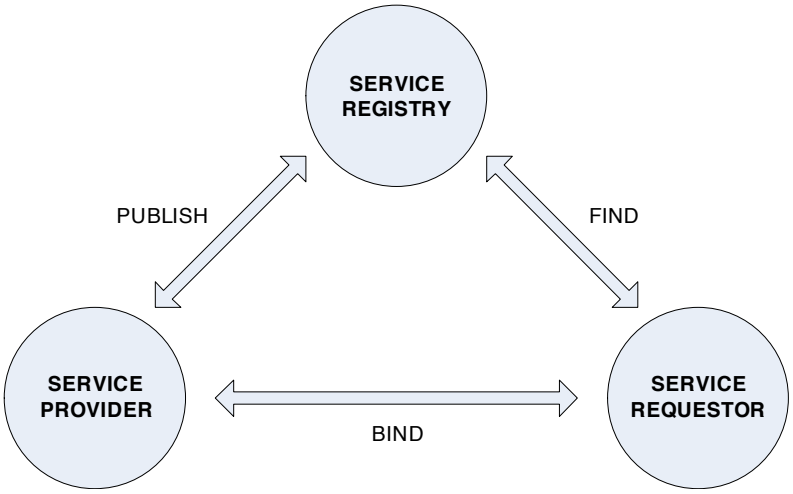

Figure 2.6: Web Service basic architecture

In mobile environments, services are the functions that networks and mobile applications make available to the user. Availability of services as well as their behavior may be subject to a client's execution environment. In case a service-oriented architecture is completely defined on a logical level, cf. [Sal02], the actual deployment of services onto target components can be left to the middleware, provisioning the services and the providing required service registration and communication links. This is possible because standards exist for service registration, discovery and invocation.

Our service-oriented approach supports the specification of deployment architectures and enables a mapping of logical service and component architectures to these deployment architectures. Beyond that, we will not be focusing further on deployment and operations activities.

2.3 Service-Oriented Architectures

The term Service-Oriented Architecture (SOA) has many definitions, cf. [Bar06, Art06, Erl04]. Most definitions agree that it relates to a set of services that can be understood, designed, implemented and operated as part of a system. Services in such an architecture are provided by system components and have defined interfaces. Service-oriented development approaches and technologies support the development of SOAs. In the mid 2000s, a substantial "hype" emerged around everything SOA, contributing to widespread adoption and large quantities of research. The hype subsided and not all quickly initiated development projects were successful, but the underlying principles and technologies have gained strong momentum in the industry and keep influencing academic work, cf. REST-style Web APIs and Microservices architectures as described below. Services and supporting architectures provide the foundation for today's Internet applications, mobile architectures and cloud based systems.

In the remainder of this section, we describe the application of service-oriented approaches and technologies to information systems and large scale system integration, as described in the introduction. In particular, we describe the importance of Web standards and Web Services that enabled a quick and broad adoption of Service-Oriented Architectures. We use these application domains and technologies as a representative for a concrete realization of services, highlighting many of the concepts and related difficulties.

2.3.1 Web Service Architecture and Technologies

Service-oriented architectures can be supported by many technologies and communication protocols, including Remote Procedure Calls (RPC) such as in CORBA [OHE97] and DCOM [Ses97], message-oriented middleware (MOM) [BCSS99, HW03] and in particular Web Services [New02, Wal02] using the HTTP [Fie99] protocol.

Web Services middleware [PB01] and Web Service based SOAs became a major driver in innovation and technology for system integration due to their ease of use and low barriers for interoperability. The general concepts behind Web Services were not new; they were taken from open Internet and World Wide Web (WWW) standards, interface description languages (IDLs), component-oriented and message passing architectures and to some degree from technology platforms such as CORBA and the Java Enterprise Platform [SUN06]. The combination of these technologies and ideas, the foundation on easy to use open standards and support from several major players in the IT industry gave this approach big leverage. Web Services provide a way of integrating heterogeneous systems in a loosely coupled way, thus supporting the creation of stable, extensible, scalable architectures. In order to use services successfully, service architects need to design versatile, reusable, independent services that contribute to the business goals of the system and enable a loose coupling of systems composed out of components or subsystems, interoperating via Web Services.

W3C's *Web Services Architecture* [W3C04b] defines Web Services as follows:

> A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

Systems built according to the Web Services Architecture [W3C04b], often called WS-* systems, have the following entities: First, service providers, which are technical systems (agents or servers) that make available and perform services for the environment on behalf of human or business entities. The provided services have a defined description (WSD), interface and usage semantics. Second, service requesters (or consumers, users) select and use the services that satisfy their business needs. Service requesters similarly are technical systems acting on behalf of human or business entities. Third, optionally, an intermediary registry or directory may connect service requesters and service providers; it provides service registration and lookup functions to both providers and requesters. Figure 2.7 depicts these concepts.

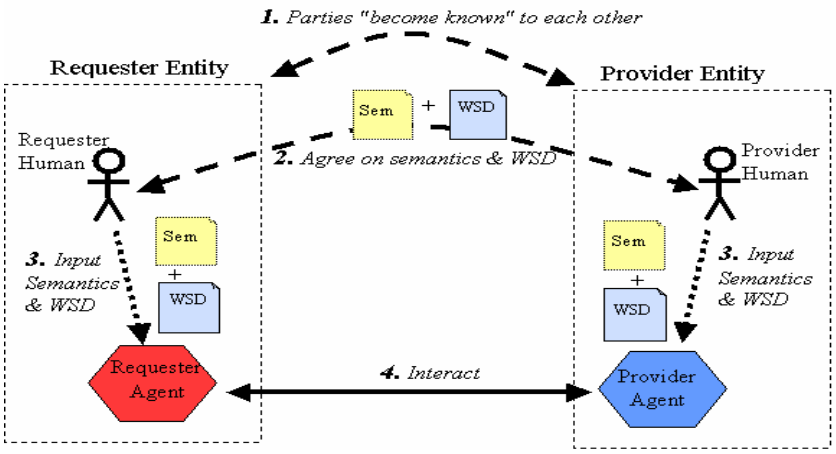

Figure 2.7: Web Service interaction, from [W3C04b]

The described pattern can be recursive; service providers can act as service requesters and thereby form higher level services by aggregating or composing other services. Depending on its nature, this step is also called service combination, choreography, or federation. An interaction between service requester and service provider is often called a conversation.

Web Service architectures focus on interactions between distributed entities exchanging documents, also called messages. A document—often in XML representation—sent by the requester to a service provider constitutes the sole input for the service provider. The service interaction is defined by the content of the exchanged documents, not by the way

the documents are transported or how they are created. This means in particular that service implementations are exchangeable as long as the exchanged documents remain the same. Due to this document-centric nature, Web Service based systems have more in common with message- passing architectures than with distributed object communication architectures. The real intention of Web Services is the loose coupling of asynchronous processes based on simple, document-centric designs, cf. [Vog03].

Several development environments and middleware solutions exist for the development and operation of SOAs, such as Microsoft's DotNET [PB01] and Java's Enterprise Edition [SUN06] with its Glassfish reference implementation [Ora13]. The emergence of these frameworks contributed to the widespread adoption of SOAs. Tool support automates development of such systems in unprecedented ways. Developers can define, deploy, publish and access existing component functionality through Web Services via Internet protocols in a matter of minutes. It is possible to wrap public components of traditional CORBA, RMI [SUN06] and DCOM based distributed object systems using Web Services, accessible over HTTP Internet protocols in SOAP XML representation. This allows for easy interoperability and system integration, but when applied to fine-grained distributed objects may lead to overly complex, underperforming systems rich with RPC style interactions. This is one of the reasons why many early SOA efforts failed in practice. Traditionally trained system architects need to adjust how to structure system architectures; services are more than a wrapper layer on top of existing components and technologies with the goal of enabling interoperability. The work in this thesis will help to design better service-oriented architectures.

2.3.2 Web Service Description and Semantics

One of the key elements of SOAs is the availability of service interface descriptions in computer interpretable formats. Web Services are described following W3C's WSDL [W3C01] specification. It is an XML based language for describing network services, providing a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations [Bar06] and messages are described in an abstract way, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services). WSDL is extensible to allow description of endpoints and their messages regardless of what message formats or network protocols are used to communicate, cf. [W3C01].

Web Service descriptions can be published in service registries and directories. UDDI (Universal Description, Discovery and Integration, cf. [OAS05b] and ebXML (Electronic Business using eXtensible Markup Language, cf. [OAS05a] define standards and exchange formats for accessing Web Service registries. Service consumers can then discover services based on their descriptions and additional semantic information.

Real world concepts can be expressed in form of ontologies, for instance in W3C's OWL (Web Ontology Language) [OWL04a] language, based on the RDF (Resource Description

Framework) [W3C04a] and XML. Ontologies can be annotated to service descriptions. One of the practical challenges of course is to find a suitable ontology for a service and to create meaningful associations to the numerous concepts within the ontology. The Semantic Web [BLHL+01] is a vision for a future Web, where information will have associated meaning rather than being pure representation. This enables automatic processing and interpretation of Web contents. OWL-S [OWL04b] is an ontology to describe Web Service properties and capabilities in an effort to create Semantic Web Services (SWS) [MSZ01]. The goal is to automate Web Service description, publishing and discovery, as well as Web Service composition and orchestration. OWL-S provides a precise description of the service, its usage interface and its behavior in form of a service profile and process model under the umbrella of an upper ontology. An alternative approach to associate semantics with Web Service descriptions is W3C's WSDL-S [W3C05] seamlessly extending WSDL and independent from any form of semantic ontology representation. It supports OWL-S as a semantic model. Further work on Web Services semantics includes the Web Service Modeling Ontology (WSMO) [RKL+05], attempting to automate all tasks related to the management of Web Services, the Web Service Modeling Framework (WSMF) [FB02], expressing service capabilities, and the Web Service Modeling Language (WSML) [dBLK+05], providing a logics-based representation for the WSMF.

Web Service semantics standards and approaches are a way of providing more precise, potentially formal specifications of services, usable in design and operational contexts. As such, they are comparable with our goals of providing a seamless development approach for services based on a formal system model. Semantics approaches only target the meaning of software artifacts in relation to one another and to the real world. Our approach enables a refinement of service interfaces into actually implementable component specifications and provides a more flexible, powerful way of reasoning about system behavior.

2.3.3 Web Service Development Methodology

Substantial work exists for how to apply Web Services systematically during development. [ZKG04] point out the necessity for a service-oriented analysis and development approach, covering several levels of abstraction. These include an "Application Domain" describing object-oriented operations cf. [Jac92, Boo93], an "Architecture Domain" grouping sets of operations logically as services and a "Business Domain", providing sequencing, selection, and execution of services—also known as choreography—resulting in business processes. Of particular interest are various forms of Web Service composition, enabling the definition of larger services or subsystems out of individual services. The following list provides Web Service methodology exemplars.

- *Web Services Choreography Interface (WSCI)* [W3C02] is a W3C standard that defines the dynamic interface behavior of Web Service interactions based on the static interface descriptions—as for example defined in WSDL—of the partaking Web Ser-

vices. WSCI describes the flow of messages connecting Web Services in an interaction, covering message sequences, exception behavior and transaction boundaries.

- *WS-BPEL* [OAS06] is an OASIS standard for the specification of business process execution based on Web Services. It provides means to express behavior sequences, exceptional and recovery behavior among the participating roles in business processes. "Concrete specifications" describe the behavior of participants of business processes explicitly, while "abstract specifications" only prescribe message exchange.

- *Web Services Flow Language (WSFL)* [Ley01] is an XML-based language by IBM describing Web Service compositions through usage and interaction patterns. It is intended to be layered on top of the Web Service description layer (WSDL). Usage patterns describe how to use a collection of Web Services in order to achieve business goals. They specify execution sequences, data and control flow among Web Services. Interaction patterns define the detailed interactions of Web Services from the viewpoint of individual services. This decentralized form of describing Web Service compositions can be used to form system wide models.

- *Self-Serv* [BSD03] is an approach to provide a framework for Web Service composition and orchestration that relies on Peer-to-Peer (P2P) mechanisms. New services can be composed declaratively of existing ones using multiple service attributes as selection criteria. Web service orchestration takes palace dynamically without a central coordinating entity by means of the framework's service containers and the concept of composite services.

The authors of [vdA03] compare different Web Service approaches according to supported patterns relevant for the coordination and composition of Web Services. One of their conclusions is that too many standards and technologies are existing, mostly influenced by specific vendor products and proprietary technologies. There are approaches targeted at generalizing service behavior specifications. XPDL [Sha02], for instance, is a technology to describe business processes using XML, independent of any implementation technologies. Its provides a high level of abstraction but falls short due to a lower precision of its definition.

The work presented in this thesis attempts to provide a generalized development approach for service-oriented architectures applicable to multiple application domains and independent of any specific implementation technology or standard. We provide a comprehensive process that covers all basic system engineering activities and at the same time provides a formal model describing distributed system structure and behavior, cf. [BKM07]. We do not provide elaborate machine readable XML specifications for model views; given the precise definition of all models, such textual languages can be defined as needed.

2.3.4 Representational State Transfer And Light-Weight Services

The prevalence of XML and WS-* technologies has been decreasing significantly over the last decade and other, more light-weight technologies have taken their place. Representational State Transfer (REST) [Fie00, BB08] provides an architectural style first proposed by Fielding, with "a coordinated set of constraints applied to the design of components in a distributed hypermedia system that can lead to a more performant and maintainable architecture" [Fie00]. Some of the basic constraints of REST include the availability of resources on the Web using defined addresses and their manipulation using meaningful well-known verbs such as GET, PUT, POST and DELETE by exchanging representations of the resource. REST calls are stateless client-server interactions that support caching. The simplicity of REST-style Web APIs contributed to their wide-spread adoption, in particular when combined with light-weight object representation formats such as JSON [JSO15] transported over the HTTP protocol.

There exist a multitude of REST-style Web APIs, supporting easy access and interoperability. Most of these APIs are documented in human readable form and require specific integration. While it is not difficult for a developer to understand and access a single REST API, it may become cumbersome to deal with the idiosyncrasies of a number of different APIs. Compared to WS-*, there lacks a machine readable standardized form of the interface. Efforts exist, however, to add back some of this, for instance in form of Swagger[2] or of JSON-LD with Hydra [LG12, Lan14]. JSON-LD, in particular, adds hyperlinks to JSON responses from REST APIs, and a defined vocabulary to describe the interface itself.

To apply service orchestration to REST services, work exists to extend WS-BPEL with concepts of resource and resource manipulation through standard operations GET, PUT, POST and DELETE, cf. [Pau09]. This enables to define composite business processes out of WS-* and REST services.

2.3.5 Microservices

Microservices [New15] is an architectural pattern for service-oriented architectures. The term Microservices emerged in the recent years and has become widely popular, as a recognizable name for a pattern that has been existing since some time, cf. domain-driven design [Eva03], strengthened by recent developments in virtualization and infrastructure automation among others. The basic idea is to compose larger-scale systems of relatively small, maximally independent service components that can be separately deployed and that manage their own persistence. There is no dependency between services other than the service call interface. In particular, there are no dependencies on shared persistence layers (e.g. database management systems) and on shared services middleware such as an Enterprise Service Bus. Services are typically designed around the individual business

[2] `http://swagger.io/`

capabilities of an organization to accomplish maximal modularity. Often, services are exposed via REST-style Web APIs. Internet technology companies such as Netflix[3] and Amazon[4] made this architectural pattern popular.

Among the advantages of a Microservices architecture is the possibility to independently deploy and manage the services. A new version of a service can be rolled out completely separate from the rest of the system, as long as the service interface remains unchanged. Development within organizations can be managed much more flexibly, with full stack development teams focused completely on one service, without the need of crosscutting teams such as front-end development, back-end development, operations, QA, and product management endlessly coordinating on releasing coherent suites of components or services within a "monolith" architecture. The recent emergence of cloud-based, virtualized compute infrastructure, of containerized deployment technologies such as Docker[5], and of infrastructure automation such as with Puppet[6] and Chef[7] make it feasible to quickly test, deploy to production and upgrade independent service components within minutes. Other advantages include the ability to choose the most suitable technology platform for each service instead of being forced on a common "one size fits all" platform.

Potential disadvantages of a Microservices architecture include increased system complexity due to a larger degree of distribution and concurrency, the necessity to design service boundaries and interfaces well before implementing the services to maintain service modularity, and the need for high infrastructure automation and a proficient DevOps mentality in the development team. Microservices are most likely not the best option for a small development effort managed by a small development team. On the other hand, larger organizations upwards of 30 developers and large-scale systems can strongly benefit from such an approach, as proven by the before mentioned companies.

2.4 Systematic and Effective Development With Processes

In this section, we describe the importance of process for the development of distributed systems. Development processes are mechanisms to make system engineering more systematic. Describing the stages of system development from conceptualization to development and beyond requires the coordination of activities over an extended period of time. Core processes include project management, quality assurance and system engineering. Work results—also called work products or artifacts—are created along the way. Keeping work products consistent in projects of anything larger than trivial size and duration requires

[3] http://www.netflix.com/
[4] http://www.amazon.com/
[5] https://www.docker.com/
[6] https://puppetlabs.com/
[7] https://www.chef.io/chef/

discipline and systematic steps. The ability to trace from requirements to functions of the delivered system is an important part of a project's customer–supplier relationship. Missing or inconsistent work products can compromise system quality and project productivity.

In the following, we will explain basic concepts and terminology for development processes, and introduce exemplar development process models. We will reference these in subsequent chapters when introducing our approach.

2.4.1 Purpose and Benefits of Development Processes

Figure 2.8 illustrates essential concepts of a process, showing part of a hypothetical process. It shows the activities of design, implementation and test during system development in relation to select work products required and produced by these activities. Activities, depicted as ovals, represent work performed by individuals. Work products, depicted as rectangular shapes, represent the artifacts used or created by individuals while performing activities. Input/output dependencies between activities and work products are depicted as arrows labeled "Product Flow"—for instance "Component Specification" as input for the "Implementation" activity, resulting in a "Component Implementation"". Consistency dependencies between work products are depicted as arrows labeled "Product Consistency Dependency", for instance the statement that a specific "Component Implementation"" must "implement" its corresponding "Component Specification".

Managing the complexity of design and implementation artifacts is particularly important for distributed systems. The best specification has greatly diminished value if it cannot be transformed systematically into a design and implementation. Systematic in this case refers to progressing in defined, small steps, potentially coordinating multiple individuals contributing to defined work products. An *artifact model* is often at the center of a development process in order to describe work products and their dependencies, so that work products can be kept consistent. Quality can be "constructed" into a system by following systematic, repeatable work steps laid out in a process. This is complemented by "analytic" quality assurance activities, such a testing or document inspection.

A suitable process must describe efficient work steps that can be performed by responsible roles in a project. Doing the right thing at the right time is essential for project productivity, enabling individuals to be most efficient, keeping development cost at bay. A good process provides detailed and specific descriptions of work steps, and guidance for the use of tools and methods. Project success is further determined by the development of effective schedules and plans, and by efficient communication between project members; both can be facilitated through a process. The existence of a process increases predictability of various project variables, including time to completion, cost, and outcome quality; this supports repeatability of the development effort. Similar projects within an organization can use collected metrics as basis-of-estimates for planning of future efforts and can benefit from lessons learned.

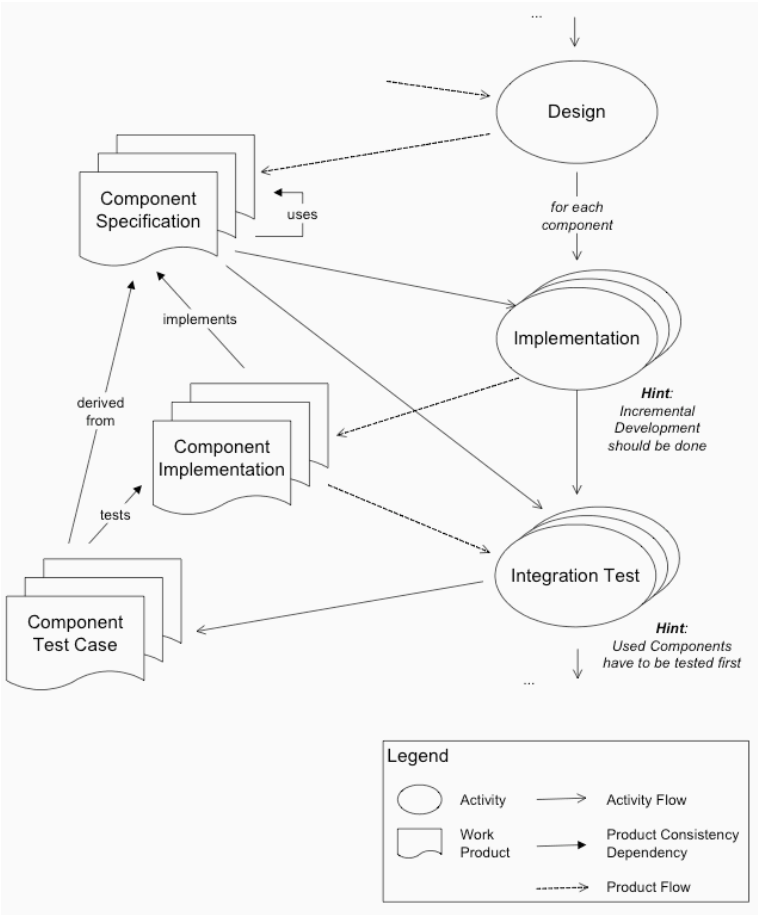

Figure 2.8: Development process activities and work products, from [GDMR04]

In case an existing process description is applied, it needs to be "tailored" to the project's and organization's specific needs and context. This may require trimming unnecessary parts of the process or simplifying process steps. It is also beneficial to add detailed method and tool guidance specific to the system under development as part of this adaptation. We open speak of *process frameworks* or *process models* for comprehensive collections of processes covering all system engineering and project management activities, applying a common set of core principles and uniform process definitions.

In summary, a suitable, well tailored development process can provide the following benefits to a project:

- Systematic, repeatable project execution with clearly defined responsibilities,
- Break-down of technical and organizational complexity into manageable pieces,
- Comparability of results and collected metrics across similar projects,
- Repeatability of result quality and project productivity for similar systems,

- Increased quality and usefulness of documentation artifacts,
- An efficient way to select and train project staff,
- Guidance for applying specific methods and tools, and for creating specific work products,
- Guidance for making project decisions and for managing critical project conditions,
- Increased degree of communication on all levels of the project, and
- A higher degree of confidence in the success and outcome of the project.

Not all processes and process frameworks provide all of these benefits equally well. Certain processes, for instance, were designed to be concise and abstract providing merely more than a project management framework—without delivering detailed activity and tool guidance. Additionally, not all existing processes can be applied to all types of systems, limiting available choice. A potential adjustment of an existing process to the system's application domain may come with a learning curve and lead to increased project overhead and less efficient development for .

2.4.2 Types of Processes

Many types of process descriptions exist, with varying background, purpose, and prerequisites. The following list provides a rough categorization, from most abstract to most concrete:

- *Process standards and process capability models:* Provide requirements for compliant processes rather than specific process definitions. Such standards and capability models may mandate the existence of various organization and project level processes. This ensures a broad coverage of activities performed, lowering the risk of missing relevant management and system engineering activities. Examples include ISO/IEC 15288 [ISO08c], ISO/IEC 12207 [ISO08b], CMMI [CMM06] and SPICE (ISO/IEC 15504) [ISO04].
- *Quality and documentation standards:* Specify elements of a development process, artifacts of process execution, and elements of the system and supporting systems that need to be documented together with the form of documentation, in order to ensure outcomes of high quality and accessibility. A prominent example is ISO 9001 [ISO08a].
- *Process models and frameworks:* Specify often generic system and/or software development processes that prescribe work products, activities and responsibilities throughout the software development cycle. Such process models and frameworks need to be tailored to specific organization and project, and can be extended with detailed method and tool guidance. Examples include the V-Modell XT [Bun12], the RUP [Kru00a] and the Unified Process [JBR99].

- *Agile methods:* Describing methods and best practices for developing systems iteratively and flexible to change, often with limited documentation and management overhead. Some agile methods realize light-weight processes. Examples include Extreme Programming (XP) [BA00] and SCRUM.
- *Workflows and procedures:* Describe specialized processes, such as for contract management, procurement, and status reporting, often tailored to an organization and tool environment. Large numbers of these workflows and procedures exist in any given organization or project—for instance mandated by a contract with a stakeholder—that must be embedded when applying a more comprehensive process model, often requiring organization specific tailoring of the process model.
- *Engineering methods:* Provide detailed guidance for applying a notation or tool as concrete solution for a specific system development step. The Unified Modeling Language (UML) [OMG11b] is an example of a method. The RUP [Kru00a] is a process framework with substantial method guidance.

It is often a challenge to integrate all required processes for a project into a consistent process landscape without creating inconsistent work products or redundant work. More flexible process standards and process models allow for an integration of a wide variety of specialized processes under a common framework.

The approach described within this thesis provides an abstract process model describing activities, work products and their dependencies independent of any specific application domain, technology or existing process description. We embed specialized engineering methods for service-oriented development of distributed system architectures into this abstract process model. Subsequently, we show a placement of this abstract process within an existing generic process model, the V-Modell XT.

2.4.3 Tailoring and Applying Processes

Applying an abstract process description to a specific system development project requires several steps and management decisions. In [GDMR04], we explain how a system development process framework, a specific development process, and the project plan for a specific project relate. Figure 2.9 illustrates these relationships. [Gna05] provides a detailed formalization and analysis of the complex dependencies between process models and specific project plans.

A *system development process framework* provides a comprehensive foundation for defining a development process specialized to an organization or a project. In general, it provides a collection of work product, activity and role definitions. Roles describe profiles of responsibility and capability that can be assigned to individuals. The dependencies between work products and activities can be expressed in various ways, for instance by explicitly linking work products to activities as required input products and produced output products. If

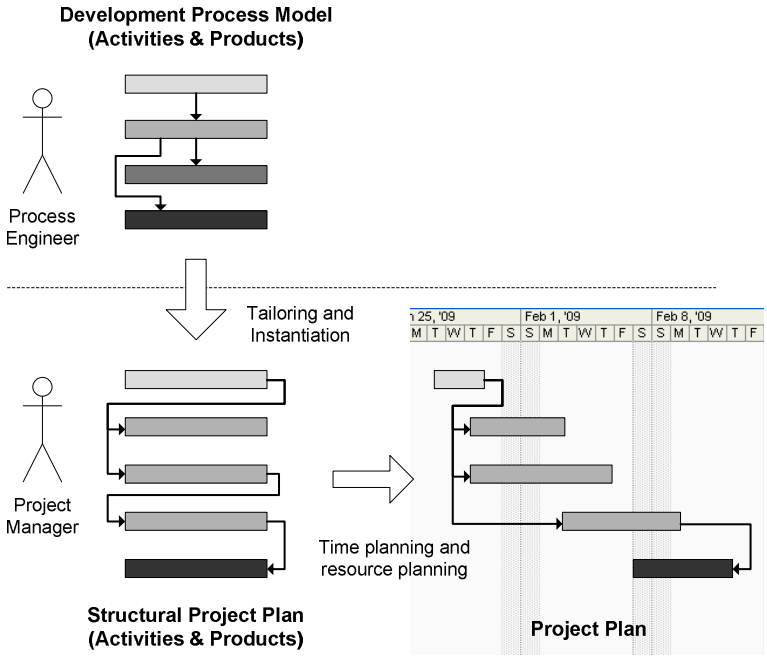

Figure 2.9: From process models to process plans, from [GDMR04]

existing, product consistency rules prescribe structure, content and linkage of produced artifacts, for instance, a constraint mandating that all requirements be traceable to component design specifications. A product state model may specify maturity states for work products. Such a state model may for instance allow a work product to be inconsistent while in "draft" state, require it to be consistent in "candidate" state, and mandate formal change control while in "released" state.

The activity of adapting a process model to a project or an organization is often called *tailoring*. Tailoring adapts the process model to the specific purpose, while maintaining internal process consistency. It may be required to modify and trim elements of the process; in this case changes are subject to a review for consistency and compliance with the intent of the process model.

The execution of a project by following a tailored development process is called an *instantiation* of the process. The artifact describing the plan for the execution is the *project plan*[8]. The project plan must follow the structures of the process; it lists all scheduled activities (also known as tasks) and expected work products (also known as deliverables or milestones). It captures dependencies between tasks and deliverables, arranges them in time, assigns effort to tasks and assigns tasks to individuals fulfilling roles. A fully fleshed out project plan is called a "resource loaded schedule". Project plans may consist of a high-level plan covering the duration of a project, refined during planning cycles for upcoming

[8]Often also called Integrated Master Schedule (IMS) or Integrated Master Plan (IMP)

iterations with detailed tasks and resource allocations. This benefits incremental development of systems where the increments are largely defined in functional scope, duration and budget. Detailed designs and feature lists can then be adjusted for each increment. The high-level plan is often protected by formal change control. Detailed plans for the next increments can be developed in a timely manner, taking specific feature requirements, user feedback, and project performance metrics to date into account. An advantage of this approach is that detailed plans only cover work increments that are short and defined enough to be effectively designed and planned out. Necessary adjustments to team productivity, requirements and system design can be incorporated during the subsequent planning interval. Following this approach, it becomes possible to embed light-weight processes and agile methods within larger plan-driven processes.

The basic structures of a project plan include the work breakdown structure (WBS) and the project execution plan. The WBS decomposes the system into manageable parts—such as subsystems and components—that can be developed, acquired and subcontracted with limited dependency on other parts. The project execution plan defines the number of releases, development cycles, and major integration steps throughout the execution time of a project. Our work in [GDMR04] provides details for relating work and execution breakdowns to defined process elements.

2.4.4 Process Descriptions

When describing processes, it is useful to distinguish several levels of abstraction. Our work in [GDMR04] suggests three levels of abstraction, depicted in Figure 2.10. The "Metamodel Layer" describes the specification language for development processes and the constraints that apply when defining a development process. The "Model Layer" defines actual processes and constraints that apply when instantiating a process for a specific project. The process elements are described using the definitions of the metamodel layer and all metamodel constraints apply (e.g. each activity must have one or more outputs products). Common process models and process frameworks are defined on the model layer. They represent classes of processes that can be adapted into specific processes by modification or pruning, while respecting the metamodel layer constraints. The most concrete layer is the "Project Layer". It contains the project plan with work breakdown structure and specific projects execution constraints. All constraints of the upper layers must be respected.

These layers are aligned with the layer model provided by the Meta Object Facility (MOF) specification [OMG06]. There, the layer of instances is identified as M0, corresponding to our "Project Layer". A language or model describing these instances is located on layer M1, corresponding to our "Model Layer". Further languages or "metamodels" describing the model elements are on next higher layers, such as our "Metamodel Layer". Because even the metamodel needs a language to be specified in, the succession of layers can be

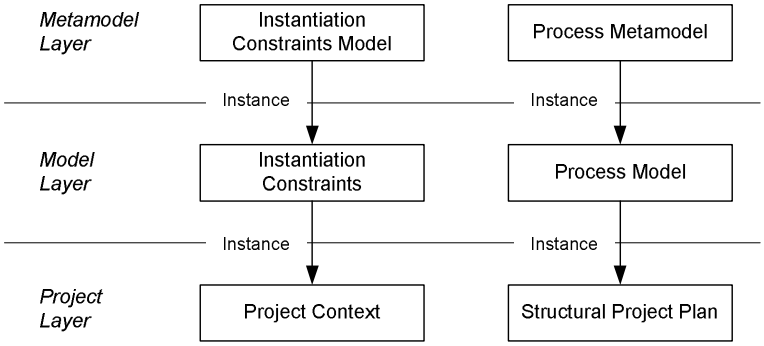

Figure 2.10: Process description metamodel, model and project layers, from [GDMR04]

considerable. The MOF lends itself as common base language for the specification of any metamodels and models. It is for instance applied in the definition of the UML2 [OMG11b].

In general, metamodels provide formal models and specifications for domain-specific modeling languages [NSKB03]. A metamodel enables and enforces the creation of models following the syntax and semantics prescribed by the metamodel. This is a requirement for generic tool support, cf. [KSLB03]. The authors claim a reduction in development cost due to the reuse of model components in the design process. The Generic Modeling Environment (GME) [KSLB03, GME06] is an exemplar toolset following this strategy.

Not all process descriptions are equally effective. To a certain degree this is determined by the level of detail in a process or domain applicability. In general, some basic requirements exist for effective process descriptions, including:

- lucid and precise definitions of the process elements, including work products, activities, roles, and dependencies, and how they relate when applying the process,
- a language to describe the process elements and their dependencies precisely,
- a language to capture dependencies systematically,
- a mechanism to tailor the process to a specific organizational or project context,
- definitions that capture and express best practices and good engineering standards,
- definitions that express and ensure compliance with selected standards, such as system life-cycle models,
- a process definition that is easy to understand and apply in concrete situations with little overhead, and
- a process that can be automated and tool supported.

2.4.5 Process Standards

The most abstract descriptions of processes exist in form of system life-cycle models and process standards. These are often official regulations developed by national and international standards bodies that impose requirements on processes and organizations. In addition to binding requirements, such standards might provide recommendations and guidelines. Two major representatives include the international standards ISO/IEC 15288 "System Life Cycle Processes" and its specialization for software systems ISO/IEC 12207 "Software Life Cycle Processes". Both have origins in the software engineering world, such as standards EIA-632 and DoD MIL-STD-498. In the following, we describe some elements of ISO/IEC 15288 that are of general interest to put the definitions and discussions in the remainder of this thesis into context.

ISO/IEC 15288 [ISO08c, Nor00] describes the life-cycle of any man-made system by defining terminology, processes, work products and activities covering conception to retirement of systems. The standard was developed and adopted by multiple international standards organizations, including ISO, IEC, and IEEE, most recently revised in 2008. The standard applies uniformly to all systems composed of software, hardware, data, human and process elements and provides harmonized terminology and process descriptions for traditionally separate fields of engineering. The standard explicitly covers the integration of software and states that "when a system element is software, the software life-cycle processes documented in ISO/IEC 12207 [ISO08b, Nor00] may be used to implement that system element." [ISO08c].

Figure 2.11 shows the 25 processes covering the life-cycle of the *system-of-interest*, grouped into enterprise, agreement, project and technical processes. Of these processes, 11 are technical. Processes exist at the level of projects, organizations and multiple organizations. Compliant organizations may be required to implemented some or all of the processes. The standard also defines 123 outcomes and 208 activities. Activities are the structural parts of processes and outcomes are work products that are the result of the successful completion of activities and processes. A special process exists describing *Tailoring* of processes to the system, project and organization. The standard supports a recursive applicability of the processes. The system-of-interest may be composed of multiple systems, which themselves may be composed of systems and system elements. Standard processes apply to systems or system elements respectively.

ISO/IEC 15288 covers the system life-cycle stages of "Concept", "Development", "Production", "Utilization", "Support" and "Retirement". The progression through these stages is controlled by decision gates, which may result in regression of state and project termination if necessary. Technical processes occur continuously during any life-cycle stage with varying emphasis.

There are further representatives in the family of software and system engineering process standards. ISO/IEC 15504 [ISO04], also known as SPICE (Software Process Improvement and Capability Determination), is a maturity model for software and system engineering

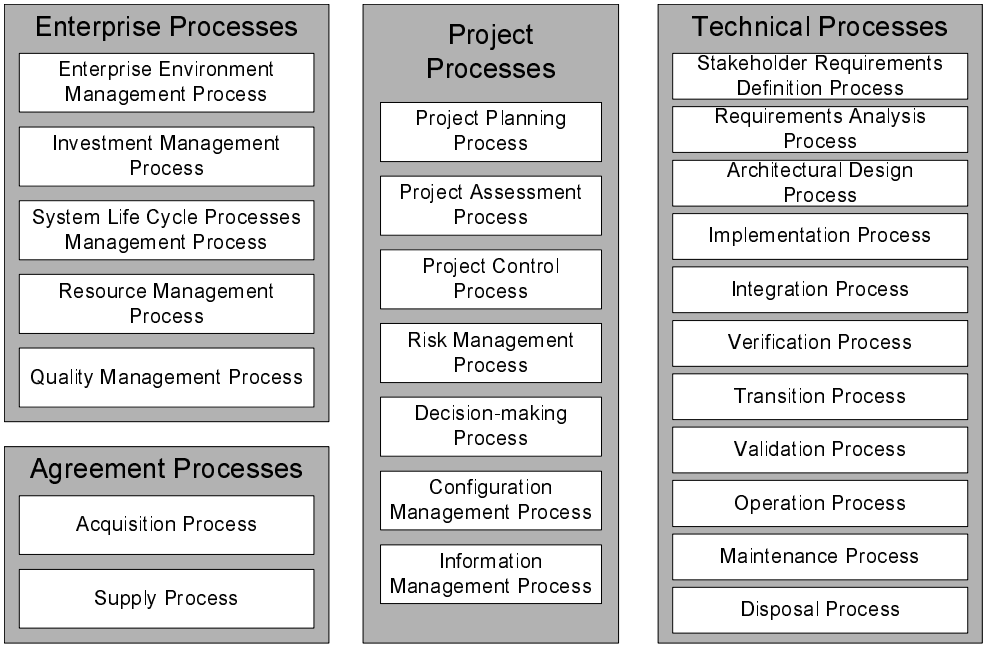

Figure 2.11: ISO/IEC 15288 System life cycle processes, from [ISO08c]

processes. Maturity assessment is based on process capabilities, with maturity levels of "Incomplete process" (level 0, least mature), "Performed process", "Managed process", "Established process", "Predictable process" and "Optimizing process" (level 5, most mature).

The ISO 9001 [ISO08a] standard provides requirements for quality management. ISO/IEC 90003 provides guidelines for the application of ISO 9001 in organizations performing software and system engineering.

2.4.6 Waterfall and V-Models

Also known as "Grand Design", the *Waterfall* model is based on a sequential succession of the main system development activities. Royce described it in [Roy70] to illustrate how software engineering was done in early days and to show its many limitations. Figure 2.12 depicts the original subsequent waterfall activities, as depicted by Royce [Roy70], from system requirements to operations: As Royce and others pointed out, the waterfall model has many limitations. While the outcome of the earlier development phases is clearly defined and transitions between phases can occur relatively smoothly, the outcome of the later verification and validation activities can be unpredictable. In some situations, modifications to requirements, design and code may be required and need to be applied consistently in order to maintain the correctness of the system. Besides, the later in the process changes get introduced, the higher the cost associated with each change. Another

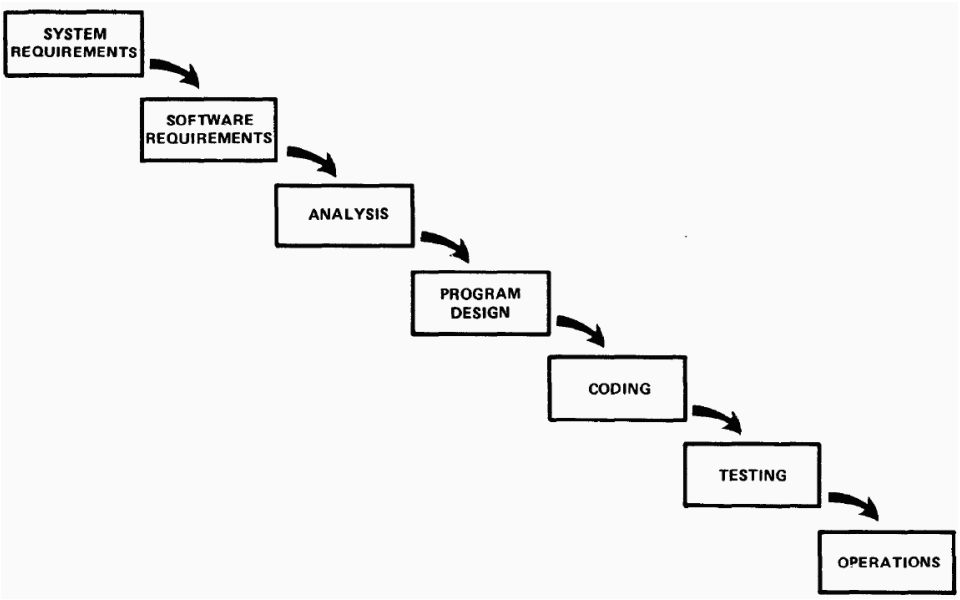

Figure 2.12: Waterfall process described by Royce, from [Roy70]

fundamental limitation of a waterfall process is that knowledge of the feasibility of specific required system features is gained incrementally during the design, implementation and verification activities. For most development efforts it is undesirable if not impossible to provide complete system requirements upfront. This limitation and the ones mentioned before render the waterfall process in its pure form rarely suitable.

"Grand Design-style" processes are still occasionally applied in development projects that justify such an approach, e.g. when requirements are known and stable and a similar system was built before. One common misconception with such processes is that change of results of completed phases is prohibited, which inevitably would lead to systems that do not fulfill the customers' needs. Instead, a defined change control process can provide a procedure for consistent changes to all artifacts, for instance to introduce new requirements during the design phase.

Numerous variants of the original waterfall exist with the following properties:

- Names and granularity of the sequential activities can differ.
- Feedback loops between adjacent phases of the development process may exist.
- Jumping back to prior phases in the waterfall may be permissible, with the mandate that subsequent phases be completed again.
- Systems are developed in increments with one "waterfall" each, based on a prior common requirements specification.
- Prototyping may occur during the design activity.

So called "V" models are commonly referred to in system engineering literature. These are processes that reflect the core waterfall phases, arranged in a form that resembles the letter "V". Requirements, system design and implementation phases flow down the left leg of the "V", while integration, verification and validation phases climb up the right leg of the "V". One of the fundamental ideas of this model is that the verification/validation of an implementation artifact directly relates to some related design artifact on the same horizontal level of the "V". For instance an implemented component needs to be verified against its component fine-design, while the fully integrated system needs to be verified against the system architecture and validated against the requirements.

In the Section 2.5, we will describe a special variant of the system engineering V-Model, the German V-Modell XT.

2.4.7 Iterative Processes

Many development processes can be applied repeatedly or flexibly for an iterative development execution. Iterative development starts with a small subset of requirements bringing system features to readiness, subsequently increasing the number of features and requirements satisfied, while incorporating lessons learned and available user and stakeholder feedback. In most cases, the main driving factors include feature prioritization and design risk. This is somewhat different from incremental development, introduced above, which focuses on a delivery of system functionality in multiple defined "releases". Often the results of prior releases are intended to remain unmodified, to save on repeated verification costs. The processes described in the following are strongly iterative in nature.

The Spiral model [Boe88] is a software development process with subsequent iterations through a defined sequence of activities, including objectives definition, risk mitigation, development and iteration planning. The opening spiral symbolizes both growing knowledge of the system and a repeated path through similar activities, indicated by the sectors the spiral transects. A significant emphasis of the spiral model is on risk analysis and mitigation.

The Unified Software Development Process [JBR99] is a process that emphasizes the execution of several concurrent activities that manipulate work products in iterations of defined length. Quality assessment and review activities ensure the consistency and completeness of critical work products at the end each iteration. A project progresses through the four phases "Inception", "Elaboration", "Construction" and "Transition", with one or multiple iterations per phase. The end of each phase represents a significant project milestone. The emphasis of individual system engineering processes changes during the various phases. During "Inception", significant effort is spent on requirements analysis and specification, while during "Construction" most emphasis is spent on implementation and verification activities.

The Rational Unified Process (RUP) [Kru00a] is a specialization of the Unified Process. The RUP includes substantial method and tool guidance; it applies the Unified Modeling Language (UML) [OMG11b] and guides the use of supporting tools. The metamodel underlying the RUP is the Software Process Engineering Metamodel (SPEM) [OMG].

2.4.8 Modular Processes

Process patterns are a concept to describe modular processes that can be composed and tailored as needed. Ambler uses process patterns to describe task-specific self-contained pieces of processes and workflows in a reusable way [Amb99]. Such patterns can be applied to solve complex tasks when needed. Störrle [Stö01] shows how process patterns can be described in great detail using the UML.

The idea of process patterns was refined by Gnatz et al. [GMP+03] resulting in a modular and extensible software development process based on collections of independent process components. These process patterns inspired the extension mechanism of the V-Modell XT, introduced below, co-developed by the same authors.

Given modular process descriptions, it becomes possible to automate the application of processes to specific development scenarios. *Process enactment* is a term for the instantiation of processes based on systematic rules, often combined with tool support to guide a project through a process. [Ost87] shows an application of *process programming*. The authors claim that process descriptions are "a form of software" [Ost87] and can benefit from methods and tools applied to software, such as configuration control, syntax checking, automatic execution etc. The authors also insist that software—and similarly process descriptions—is not "just code" and that development methods need to a apply such as design and testing.

The approach we introduced in [MRS06] leverages many similarities of software and process descriptions, for instance the existence of a process specification language, the use of configuration control to maintain revisions of the process description, the use of tools to tailor and extend the process, just to name a few. The 4everedit approach is also a descendant of process patterns as introduced in [GMP+03], and inspired the V-Modell XT tool environment.

2.5 V-Modell XT – an Extensible System Life-Cycle Process Model

The V-Modell XT [Bun12, MRD+04, RB06]—V-Modell in short—is a modern software and systems life-cycle and development process model, developed by the German government. It supports all management and technical processes that are relevant in system engineering.

It places special focus on the concept and development stages of the system life-cycle; these prepare explicitly for subsequent stages such as operations and decommissioning. We describe the V-Modell in some detail as a representative development process applicable to distributed reactive systems. Later in this thesis, we will provide a service-oriented extension of the V-Modell, referencing the fundamentals introduced here.

Processes for project management, quality assurance, configuration management, and problem/change management form the core of the V-Modell, complemented by basic technical processes, such as requirements specification, system, software and hardware development. Specialized technical and management processes exist, such as contract management, delivery and acceptance, safety and security, usability, ergonomics and others.

2.5.1 V-Modell Origins and Goals

The V-Modell XT is the mandated software and systems development process for German federal and military IT systems. The V-Modell is largely compliant with ISO/IEC standards 15288 [ISO08c] and 12207 [ISO08b] and provides mappings to other process standards, such as CMMI [CMM06] and ISO 9001 [ISO08a]. The V-Modell XT embraces the core structures of the traditional system engineering "V-Model" explained above but goes far beyond. In particular, it specifies how multiple cycles of the "V" can be combined and how system breakdown can lead to "parallel V's" for independent system elements. The V-Modell XT was released to the public under the Apache 2.0 license.

The V-Modell XT provides an easy to understand and use generic development process model that can be flexibly adapted to the needs of organizations and projects. The main objectives of the V-Modell XT are stated as the following [Bun12]:

- *Minimization of Project Risks:* The V-Modell improves project transparency and project control by providing standardized approaches, result expectations and responsible *Roles*. It enables early recognition of planning deviations and risks.
- *Improvement and Guarantee of Quality:* Project results are complete and of desired quality when following the process and easier to understand and verify due to uniform content. Defined interim results exist throughout the project stages.
- *Reduction of Project and System Life-Cycle Total Cost:* The cost for each phase of the system life-cycle can be calculated, estimated and controlled systematically; costs are comparable given the standardized process with uniform results. This reduces acquirer dependency on the supplier and enables synergies for subsequent projects.
- *Improvement of Communication between all Stakeholders:* The standardized, uniform definition of all relevant concepts and terms increases the mutual understanding between all stakeholders, reducing friction between user, acquirer, supplier and developer.

2.5.2 Basic Concepts

The V-Modell XT provides a precise metamodel defining all model concepts and their relations. Figure 2.13 shows a part of this metamodel specifying the relation between work products, activities and roles. The V-Modell XT follows this metamodel strictly, enabling powerful tool support and extensibility. The model itself is authored in XML according to an XML schema compliant with the metamodel. This enables a lossless generation of the model and its tailored variants in multiple representations such as in PDF and HTML [MRS06].

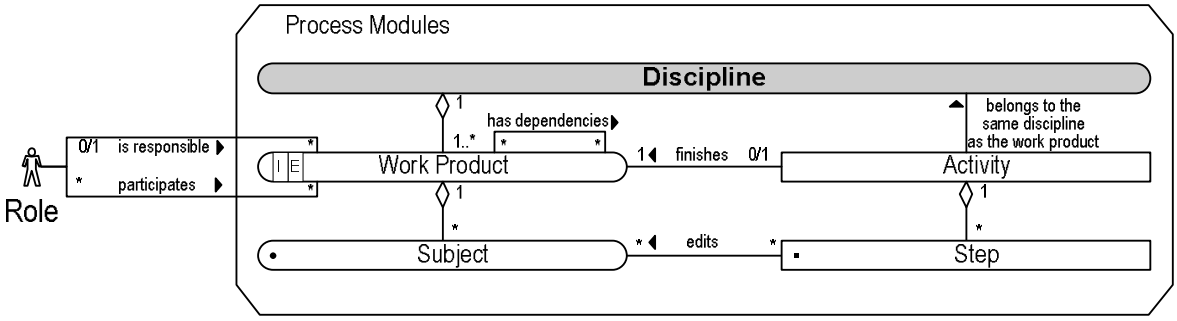

Figure 2.13: Part of V-Modell XT metamodel, from [Bun12]

The central concepts of the V-Modell include, cf. Figure 2.13:

- *Role:* Describes a profile of responsibility and required capability applicable to individuals in a project. Individuals take on roles during a the execution of a project.
- *Work Product:* Essential project outcome and artifact; can be of any representation, such as document, design model, source code, and deliverable system. Each work product has a prescribed structure and content and can be substructured into subjects. Subjects can be represented as sections in a document or as views in a design model. Each work product has exactly one role assignment that is responsible for its manipulation, and lists other roles that participate. Work products are the unit of quality control and configuration management. An evaluation specification provides a checklist for their quality.
- *Product Dependency:* Defines a consistency relation between the contents of two or more work products. Adhering to and checking product dependencies ensures that new work products in a project will be kept consistent with existing products. Product dependencies are an important means to assure product quality and to trace information across products, for instance from the requirement specification to the software architecture.
- *Activity:* Defines the actions that need to be performed in order to create and modify one work product. Activities provide guidance for the actual "work" within a project.

One activity is associated exactly with one work product. Activities can be structured into (work) steps.

- *Discipline:* Groups a number of work products and activities by topic. Work products are typically closely connected within the same discipline.
- *Project:* The organizational unit that executes a V-Modell XT process in a coordinated way. Multiple organizations and projects can be involved in the development of one system, such as an acquirer working with supplier organizations. Projects interact through the exchange of released work products.
- *Process Module:* Packages related work products, activities, roles, and other V-Modell XT elements. Each process module covers a particular project process such as project management and requirements specification, or extends another process module with modular additions to the process such as for security/safety. Process modules can be understood, applied and modified rather independently and are the units of tailoring and extension of the V-Modell XT.

The V-Modell XT provides a total of 22 process modules, depicted in Figure 2.14 with their dependencies. *Tailoring* is the activity of adapting the V-Modell XT to a specific project or organization, selecting the suitable process modules from the repository of available ones. The result is a consistent system life-cycle process that only contains necessary model elements. The smallest compliant process instance only requires the core four process modules, which are managerial and nontechnical in nature. Technical process modules complement these core modules. Some process modules exist in both acquirer and supplier variants.

2.5.3 System Engineering Processes

In the following, we focus on the basic technical processes, which are defined in these four process modules:

- *Specification of Requirements:* Preparing a requirements specification document based on a project proposal; evaluating elicited requirements in terms of effort, cost and importance.
- *System Development:* Decomposing a complex system into manageable units of software, hardware, supporting systems and additional materials (such as user and administration manuals); integrating the units into the deliverable system.
- *Software Development:* Developing an individual software unit, which includes specification, software architecture design, implementation and integration, test specification and unit evaluation.
- *Hardware Development:* Developing an individual hardware unit, which includes specification, hardware architecture design, realization and partial integration, test specification and unit evaluation.

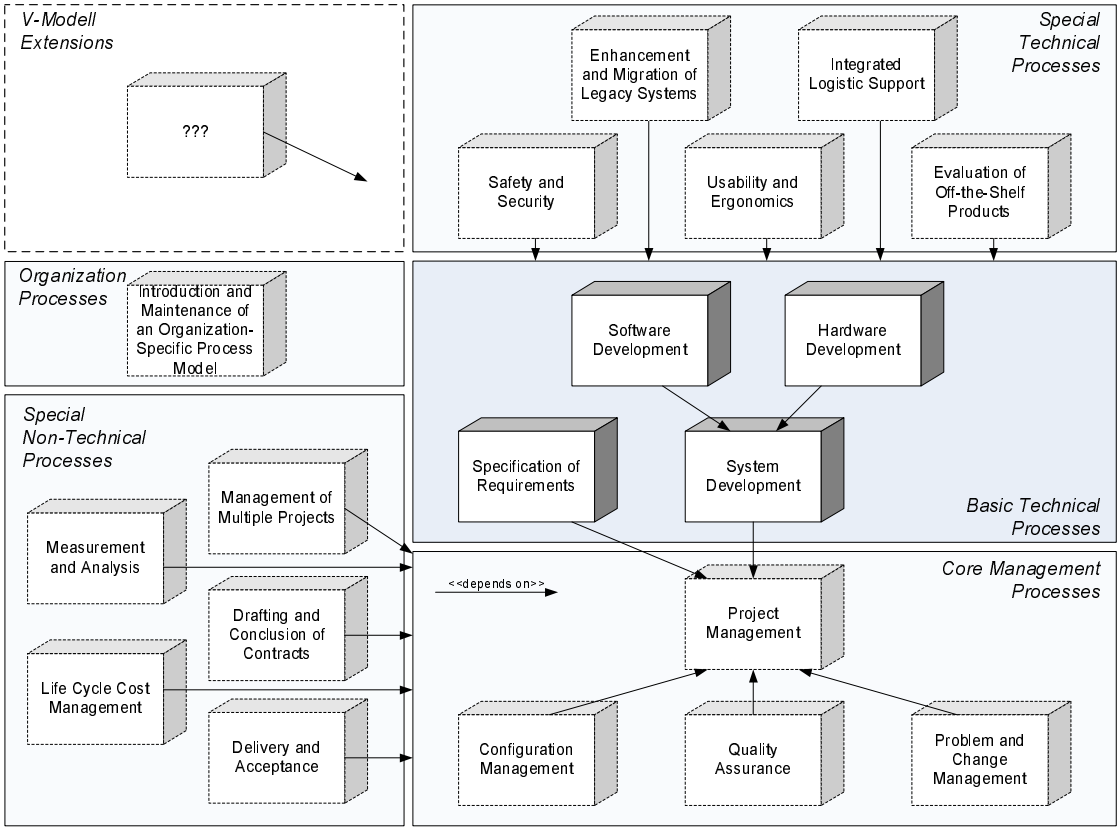

Figure 2.14: V-Modell XT process modules

The name giving system engineering "V" is represented in the V-Modell in form of the *decision gates* of the four basic engineering process modules, cf. Figure 2.15. Decision gates are significant points in a project when the project management certifies a certain degree of project completion based on the presence of required quality checked work products. Possible decision outcomes include moving ahead to the next stage, remaining in the current stage and applying corrective actions, and terminating the project.

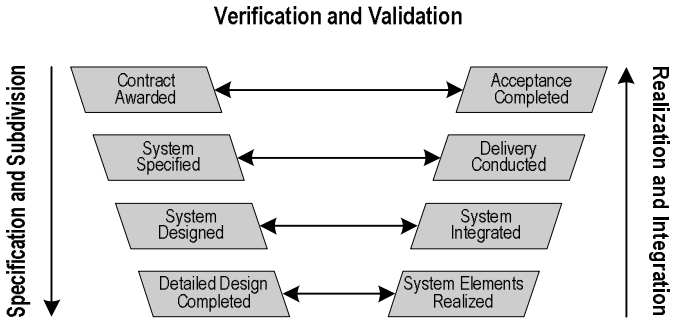

Figure 2.15: V-Modell XT decision gates during system development

An engineering project progresses through the decision gates in more or less straightforward flow. On a general level, the V-Modell does not make any statement about the order in which the decision gates are arranged, or whether repetitions and cycles are expected and possible. Figure 2.16 shows an exemplar arrangement of the decision gates for a typical engineering project with hierarchical system breakdown, progressing from the Contract Award to the Acceptance stage along the two legs of the "V". Activities and work products at the same level are related, such as the integrated system element matching its detailed design. The figure also identifies important work products verified at each of the decision gates. Lower decision gates occur once per system element—based on the hierarchical decomposition of the system—indicated by the fork/join symbols. In this case, the work products related to these decision gates exist once for each system element, such as one "System Architecture Document" per system element. Other project specializations can lead to different sequences of decision gates. Process modules can define additional decision gates, for instance preceding the Contract Award for setting up the project and getting to the contract.

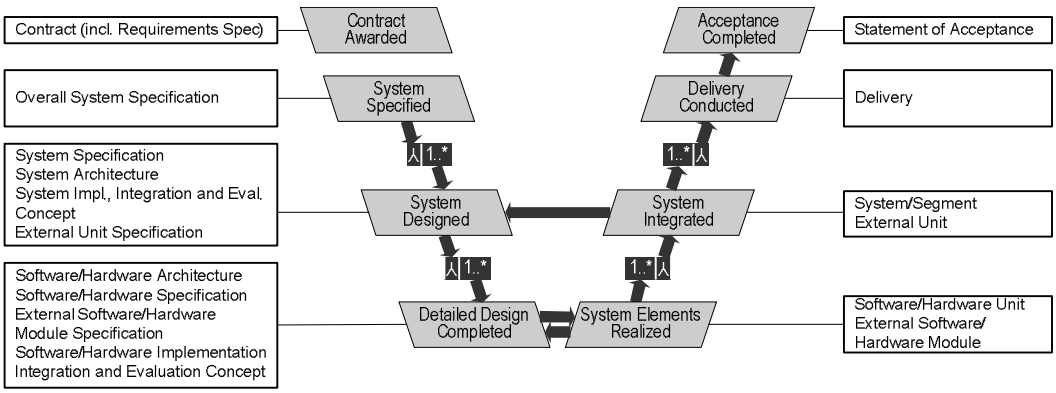

Figure 2.16: Standard arrangement of V-Modell XT decision gates with required artifacts

The V-Modell XT mirrors the hierarchical decomposition of the system. The project level defines contractual artifacts as well as the "Overall System", comprising the technical system under development and logistics elements. The actual technical system contains elements on two levels. The system level defines the system, which decompose into units, optionally arranged within segments. The system level describes the large parts of the system and their integration out of units, the lowest level elements on the system level. Units are either software or hardware units, and can get acquired externally. On the unit level, software and hardware units decompose further into components (if necessary) and modules. The V-Modell XT defines uniform artifacts for developing elements on each of these levels:

- A *Specification* describes the context and purpose of a system element and its interfaces from a black-box view, as well as any nonfunctional requirements. Additionally, any internal interfaces between subelements are documented.

- An *Architecture/Design* describes the system, software or hardware architecture of one system element, including a decomposition into subelements and interfaces.
- An *Implementation, Integration and Evaluation Concept* describes details and plans for the actual implementation, the tools and procedures used, the integration and the execution environment, as well as any test and verification strategies.
- The actual *System Element* itself, such as a Software Unit, implemented and ready for integration.

Figure 2.17 shows the core engineering artifacts and some product dependencies. The figure shows work products as rectangular shapes. The name of the artifact is printed in bold face, and important product subjects are listed below. Specifications are placed on the left, architecture/design and integration/verification artifacts in the middle and system elements on the right. Requirements can be traced through all these artifacts following the product dependency arrows.

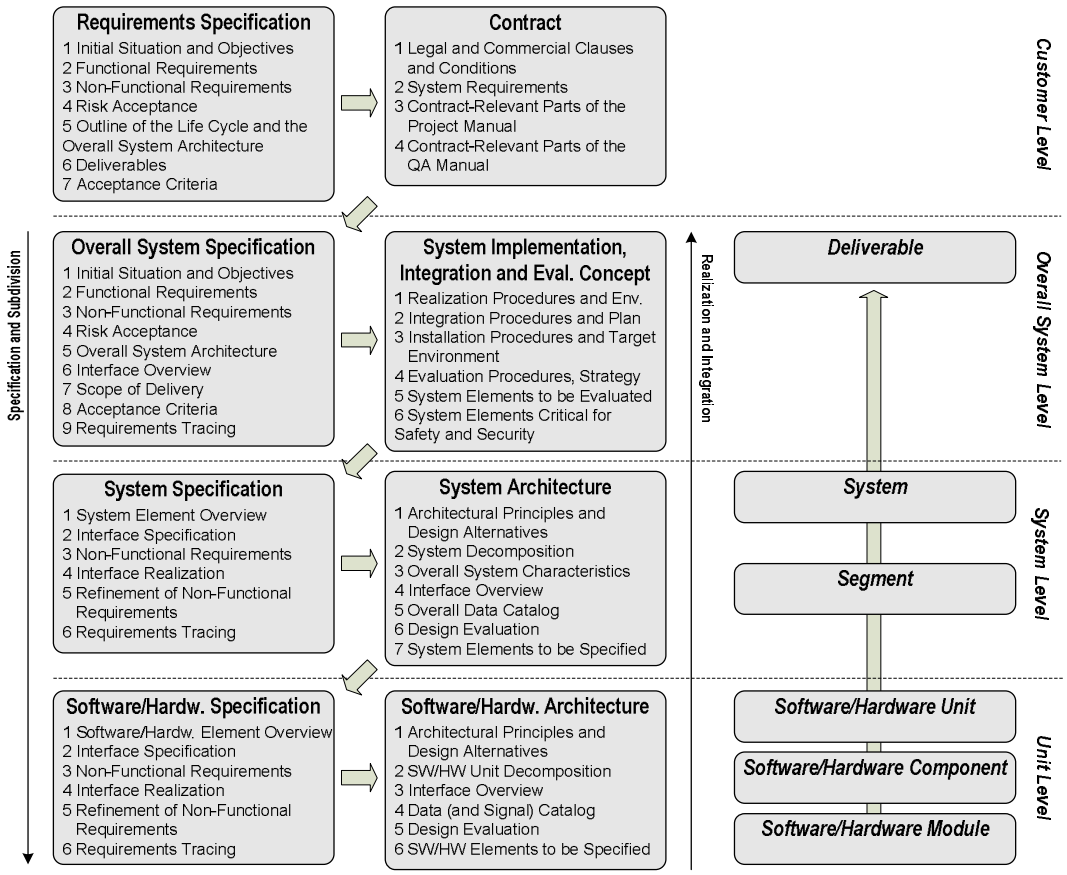

Figure 2.17: V-Modell XT system engineering artifacts

2.5.4 Developing Distributed Reactive Systems

The V-Modell is well suited to guide the development of distributed reactive systems. The need for specific process modules depends on the need and scope of the system and the development project. In many cases process modules for both software and hardware development are required. Additional modules may be selected based on project constraints, such as "Safety and Security". The basic system engineering activities of the V-Modell fully apply to distributed reactive systems. Particularly beneficial for distributed systems is the systematic decomposition of the system architecture into segments, software and hardware units, components and modules, and the matching integration and verification process.

The V-Modell, however, does not provide detailed guidance for how to manage the particular complexities within this system class. In general, it does not provide any specific method and tool guidance and only references a few potentially valuable methods and tools. This is where the extensibility of the V-Modell comes into play. Anyone can define specialized processes that integrate consistently with the existing processes on both management and engineering levels. The V-Modell provides significant support for such extensions at any level, including adding method and tool references, new work products and activities, and extensions to existing work products and activities. We will use this extension capability in Chapter 5 to add a process module for service-oriented development.

2.6 Elements of a System Development Approach

A comprehensive system development approach consists of a number of related parts. In the preceding sections, we introduced development processes as an important step to enable systematic development in larger-scale projects. Development processes rest on lower level parts, as shown in Figure 2.18. Arrows indicate how the parts of an approach depend on others, forming a layered stack. We explain the meaning of these parts in the following:

- *Concepts:* Basic and derived elements of design and implementation pertaining to the class of systems under development. Concepts, for instance, include component, service, operational mode, communication channel, data type, information message, system architecture and service hierarchy. These relate to the design and information artifacts created during the development process. All concepts and their dependencies need to be defined precisely, for instance using a rigorous formalism.
- *Foundation:* This includes mathematical definitions and formalisms needed to define concepts and higher level elements of a development approach, and often part of a comprehensive theory of systems and a system model for the class of systems under development. A rigorous foundation is essential for a precisely defined software and system development approach. We will introduce a formalism and system

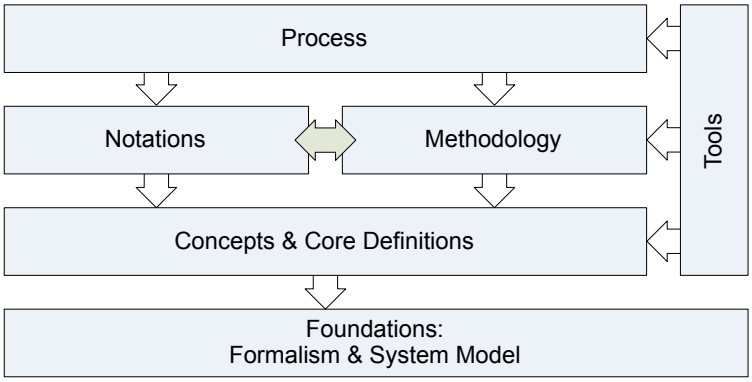

Figure 2.18: Elements of a system development approach

model in the next chapter based on the FOCUS theory and the mathematical concept of streams. A formalism can also be based on algebras of terms, formal language grammars, and languages such as temporal logic. The formalism together with core definitions provides a semantics domain. The meaning if any development approach element can be defined by providing a mapping to this semantics domain.

- *Notations:* Representations for the concepts of an approach and their relationships, for use during system development and implementation. Notations—also called description techniques—are used to specify system elements during design activities and are particularly important when using model-based development techniques. Notations need to be suitable to express the abstractions of the underlying model, and enable precise interpretation and flexible modification. Notations include textual, formal, and graphical languages with a defined syntax. Different notations can provide alternative views on the same concept.

- *Methodology:* Operations to transform and relate the concepts of an approach, as represented by notations, in order to enable systematic, iterative development. Supports starting from high-level elements and first principles all the way to completing a detailed system design and implementation. Operations include composing and decomposing components and services, performing an interface abstraction, relating the hierarchical layers of a system architecture built of components, deriving a state based behavior model from an interaction specification etc. Transformation operations are the building blocks of iterative development. More complex operations can be expressed as sequence of basic operations.

- *Tools:* Automation for repeating development steps, such as authoring a design model, tracing information between development artifacts, and maintaining versions and configurations of project artifacts. Software implementation tools include code development environments, compilers, automated build systems, deployment management tools etc. Model checkers and interactive theorem provers can assist with the verification of specifications. The stricter a tool represents the semantics of the

concepts as expressed by the formalism, the higher the degree of confidence in the correctness of the outcome.

- *Process:* Descriptions of development artifacts, activities and responsibilities and their arrangement, applied to organize larger scale development projects. This makes development efforts repeatable and helps to increase result quality while keeping development cost and schedule manageable. Processes apply an approach's notations, methodology and tools, although often defined in a generic way.

A comprehensive development approach provides all of these elements. Complex dependencies exist between such elements. For instance, an expressive graphical notation requires complex semantics definitions and an expressive formalism. A careful balance must exist between the desire to provide powerful notations, methodology and tools, and the complexity caused by proving the consistency and correctness of the overall approach. The cost of providing effective tool support and the learning curve for prospective users of an approach are additional influence factors.

2.7 Summary

In this chapter, we have shown the importance of systematic development, in particular for distributed reactive systems that are notoriously complex to develop due to the many interactions between system components. We have described the major system engineering activities including requirements engineering, system design, system implementation, integration and verification, and deployment leading to an operational system. We have highlighted the benefits that service-orientation can bring during these activities, in particular when developing service-oriented architectures. We have pointed out related work and placed our approach in the context of the literature.

We have shown how applying a development process can make development efforts repeatable and lower development cost while ensuring and increasing system quality. We have described the system engineering process of the V-Modell XT in some detail for several reasons. First, it provides a representative systematic development process covering all system engineering activities. Second, it provides processes strongly aligned with the hierarchic decomposition of the system, resulting in a scalable process particularly suited for distributed systems with their many independent components. Third, it is flexibly extensible on all levels, enabling us to embed our service-oriented approach seamlessly into the process.

The following three chapters will lead to the extension of the V-Modell XT with our service-oriented process. In the next chapter, we will provide the necessary formal underpinnings that will enable us to define our approach and generic process formally. Based on these formal definitions, we will introduce the artifact model for our approach in Chapter 4 followed by generic process descriptions in Chapter 5, concluding with the V-Modell extension.

3 A Formal Model for Service-Oriented System Development

In this chapter, we describe a comprehensive formal system model for service-oriented system development, in particular the specification of system architectures and design models. The model is based on the mathematical formalism of streams and provides constructs to specify distributed reactive systems and hierarchical system architectures. Services are represented in the model as first class elements. We describe notations applicable to such specifications and operations that manipulate them.

Contents

3.1 A Formal Model of Distributed Systems

This chapter introduces a comprehensive formal system model for service-oriented system development, focusing on the specification of system architectures and system design models. This formal model will be the basis for a subsequent definition of a development process with artifact model and activities supporting these artifacts, unfolding the semantic space established by the formal model. The process will flexibly represent concepts of the formal model for the system class we focus on in this thesis: software-intensive reactive systems with flexible but largely static system architectures made up from distributed components, suitable to support a high number of functions provided to the environment and the users.

As a starting point, we describe in this section the key elements of FOCUS [BS01], a formal description and development technique (FDDT) particularly suitable for distributed reactive systems with a static structure. It defines a simple, yet precise mathematical model based on infinite streams, and enables the formal specification of systems comprised of multiple interacting components with a defined set of communications channels between these components.

3.1.1 An Introduction to FOCUS

FOCUS is comparable in its extent and purpose to other FDDTs such as CCS [Mil89], CSP [Hoa85], TLA [Lam94], Unity [CM88] and the MSC-based approach introduced in [Krü00b]; all of these techniques support the formal specification of systems and the derivation of implementations from the specifications. In providing formal development methodology, these techniques go beyond pure formal description techniques (FDTs) such as SDL [EHS98], MSC [IT96] and Estelle [BD87]. All of the above provide formally defined syntax and semantics. This sets them apart, for instance, from the UML [OMG11b] as the as most well-known example of a semi-formal technique, cf. [BS01]. The UML offers an underlying metamodel and action-based semantics applicable to a variety of overlapping notation languages that are often applied with rather informal interpretation. We provide a more detailed discussion of related work below.

FOCUS is the foundation of the formal system model introduced throughout this chapter. We use this model to subsequently define concepts of our service-oriented development approach. Particularly relevant are the definitions enabling us to specify system structure and behavior, and to define the semantic interpretation of notations and transformation operations applied to specifications. In the following, we first introduce basic concepts and the FOCUS system model. Subsequently, we provide formal definitions. We closely follow the notational style and definitions of [Krü00b].

The FOCUS theory [BS01, BDD+92] provides a foundation for describing structure and behavior of distributed systems that are comprised of interacting components. Such components can themselves be systems and subsystems. The components and their communi-

cation channels define the structure of a system. A *component*[1] is an entity that encapsulates a state and is capable of communication and information exchange. Components exchange information exclusively though messages on communication channels. *Ports* are the endpoints of such channels on components. *Channels* are directed links between two components or between one component and the environment. Channels carry messages of designated types. Components and channels have unique names. The sets of components, channels and message types are static for a given specification. A specification represents a model of the real world that abstracts from certain details, such as implementation specific technology choices and timing variants, cf. [Bro05c]. Figure 3.1 depicts the static structure of a distributed system, taken from our running example, with four components and six named channels. The system realizes a sender, such as a station controller transceiver, communicating with a receiver component, such as a train transceiver, communicating through transport media, such as radio communications.

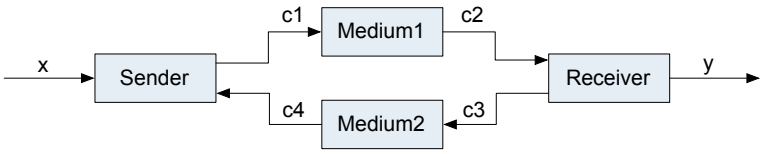

Figure 3.1: FOCUS system structure, cf. [Bro05a]

FOCUS specifies the behavior of a component as a relation of input and output messages occurring on the component's ports. More precisely, as a relation between sets of input streams and sets of output streams for all ports of a component, where a *stream* is the valuation of messages on one channel over time. Such valuations are also called *channel history* and can be of finite and infinite length. All input-output relations must be causal, meaning that output cannot depend on input that has not yet been received. FOCUS explicitly supports nondeterminism in specifications by permitting the relation of multiple outputs with one input. The FOCUS system model assumes an asynchronous, reliable, order preserving exchange of messages on channels with no time delay, implying a system wide global discrete clock. The system progresses in a sequence of global time ticks. Several messages can occur on a channel within one tick. Communication media with different properties, such as a lossy channel with a delay, must be modeled explicitly as a component.

An *architecture* is a structural view of the system showing the hierarchical decomposition of a system into interconnected components and subcomponents. FOCUS supports the definition of system architectures through its operation of component composition, thereby increasing the scalability of specifications. More complex behavior can be specified by composing individual interconnected subcomponents yielding composite components, which in

[1] In other approaches components are sometimes also called objects, modules, agents, systems and subsystems

turn can be connected and composed again, resulting in a hierarchy of components. The top level component represents the overall system. FOCUS defines a powerful notion of component *composition*, which allows for direct derivation of the syntactic and semantic interface of the composed component, given an architecture of components and their individual behavior specifications. Figure 3.2 shows the architecture of the distributed system "Communication Network" from our running example with several internal components and communication channels. The figure demonstrates how a real world system comprised of software components and communication media can be modeled. The FOCUS components from Figure 3.1 above are included within the architecture as the "Transport" component.

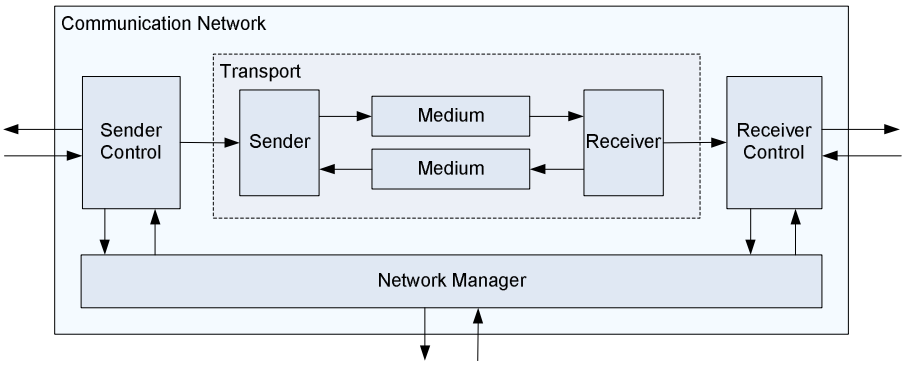

Figure 3.2: Architecture of FOCUS components

Interfaces are black-box views on component structure and interaction behavior that hide the components' inner structure and implementation. From a given architecture of components—a white-box view—it is possible to derive an interface abstraction, hiding all internal knowledge, though composition of all components of the system. Abstraction from implementation reduces the overall development complexity and increases modularity. The actual component implementation can be chosen freely as long as it respects the specification. FOCUS emphasizes interaction interfaces over white-box specifications. Figure 3.3 illustrates the result of an interface abstraction through component composition, applied to the architecture depicted in Figure 3.2. All communication channels to and from the environment remain the same, and so does the set of permissible channel valuations for these channels.

Figure 3.3: FOCUS component interface abstracted from architecture

FOCUS specifications can express different levels of time dependency. Untimed specifications are most abstract; they describe causal behavior of components exchanging information with no reference to any external timing. Such specifications are useful for distributed systems without strong timing requirements. Timed and time-synchronous specifications support the specification of time constraints for asynchronous and synchronous components respectively, and are well suited for embedded and real-time systems.

Component behavior specifications can be expressed in one of several *specification styles*, introduced in Section 3.1.3 below. FOCUS supports step-wise refinement and iterative development. It provides several concepts of *refinement*, each guaranteeing certain properties during a specification transformation step. Three types of refinement are supported: *Property refinement* occurs when all logical properties of the refined specification imply the properties of the original specification. In an input-output relation, this relates to reducing the number of outputs for the same inputs. *Interface refinement* occurs when the number of channels and types of messages are modified. *Conditional refinement* adds assumptions on the environment to both behavioral and interface refinement. We describe refinement in more detail in Section 3.3.6.

3.1.2 Basic Definitions for FOCUS and Streams

In this section, we provide basic formal definitions for FOCUS and its system model. We will use these definitions subsequently to add further formal definitions of system architecture, modeling notations and model transformation operations. Table 3.1 provides notational conventions supporting the subsequent definitions. Notational conventions and definitions are adapted from [Krü00b], which provides an extensive formal model and methodology.

The mathematical model behind FOCUS and the methodology in [Krü00b] is that of *streams*. Table 3.2 summarizes basic definitions related to streams. Streams are used in FOCUS and related approaches to specify the behavior of components, by describing the observable interactions of these components over time. Table 3.2 defines streams as finite and infinite *sequences* over a given set X. These definitions will be used to provide mathematically grounded component behavior specifications.

Table 3.2 also shows definitions extended to finite and infinite *sequences of sequences* over X. This enables to represent the concept of discrete time within streams, by assigning finite sequences of messages to each time interval of a global discrete clock.

Streams and relations on streams can be used as simple yet expressive specification technique for distributed reactive systems, cf. [BS01, Möl99, Ste97]. Stream-based formalisms are advantageous when working with predicate logic based formal specifications. They support flexible refinement notions, cf. [BS01, Kle98, Sch98]. Some approaches, such as [Krü00b] go beyond streams in component behavior definitions and associate a notion of explicit state with components. The state provides an abstraction of previously received inputs and produced outputs. This simplifies the specification of preconditions

Table 3.1: Notational conventions overview, based on [Krü00b]

Notation	Informal Definition
$\mathbb{B}$	set of booleans (the constants are true and false)
$\mathbb{N}$	set of natural numbers (including 0)
$\mathbb{N}_\infty \stackrel{\text{def}}{=} \mathbb{N} \cup \{\infty\}$	set of naturals together with their supremum (∞)
$x.f$	function application, synonym to $f(x)$
$\langle Qx : r.x : p.x \rangle$	quantification over all $p.x$ for which x satisfies the quantification range $r.x$, for $Q \in \{\forall, \exists\}$ and predicates r and p
$\langle \forall x \in \mathbb{N} :: \ldots \rangle$	synonym to $\langle \forall x : x \in \mathbb{N} : \ldots \rangle$
$\mathcal{P}(X)$	powerset of any set X
$\pi_i.y \stackrel{\text{def}}{=} y_i$	projection on the i-th element of the tuple for $i \geq 1$, given sets $Y_1, Y_2, \ldots$ for tuples $y = (y_1, y_2, \ldots) \in Y_1 \times Y_2 \times \ldots$
$[m, n]$	closed interval between $m \in \mathbb{N}_\infty$ and $n \in \mathbb{N}_\infty$; $\emptyset$ if $m > n$
$g\|_T : T \to R$	function restriction of g to the domain T, for $g : D \to R$ and $T \subseteq D$
$f \circ g$	function composition of functions $f : C \to \mathcal{P}(D)$ and $g : D \to \mathcal{P}(E)$ such that $(f \circ g)(x) = \{z \in g(y) : y \in f(x)\}$

and assumption predicates on the past input/output histories at defined points in the execution.

3.1.3 FOCUS Specification Styles

The behavior of a component is expressed in FOCUS as a function mapping input streams to sets of output streams. This specifies the observable interaction behavior of a component towards the environment. Determining and specifying such component behavior functions can be difficult for given problems. The availability of suitable specification techniques is of high importance. The FOCUS approach provides the following four basic specification styles that all translate into predicate logic expressions [BS01].

- Relational style
- Equational style
- Assumption/guarantee style
- Graphical specification style

The *relational style* directly relates input and output streams of components. It is the simplest form of FOCUS specification. Figure 3.5 below shows the specification of the "Transport" component in relational specification style as an example. In this case, the syntactic interface consists of four channels $c1\_in$, $c1\_out$, $c2\_in$, $c2\_out$ with $Type(c1\_in) =$

Table 3.2: Stream notation overview, based on [Krü00b]

Notation	Informal Definition
X^*	set of finite sequences over set X ($X^* \stackrel{\text{def}}{=} \bigcup_{i \in \mathbb{N}}([0,i] \to X)$)
X^∞	set of infinite sequences over set X ($X^\infty \stackrel{\text{def}}{=} \mathbb{N} \to X$)
X^ω	set of streams over set X ($X^\omega \stackrel{\text{def}}{=} X^* \cup X^\infty$)
$x.n$	nth element of stream x
$\lvert x \rvert$	length of stream x
$x{\downarrow}n$	prefix of length n of stream x
$x{\uparrow}n$	stream obtained from x by removing the first n elements
$x \frown x'$	concatenation of streams x and x'
$<x_1,\ x_2,\ \ldots,\ x_n>$	finite stream consisting, in this order, of the elements x_1 through x_n
$y \in x$	y appears as an element in the finite stream x, such that $\langle \exists i : 1 \leq i \leq n : x_i = y \rangle$
$<>$	the empty stream
x_1^n	the stream consisting of n consecutive copies of element x_1
$Y \textcircled{c} x$	stream obtained from x by dropping all elements not contained in Y
$x\vert_{[m,n]}$	stream obtained from x by considering only elements m through n
$s : [1,m] \to X^*$	finite timed stream over set X, for $m \in \mathbb{N}$
$s : \mathbb{N} \to X^*$	infinite timed stream over set X
$s(t)$	sequence of messages communicated at time t in the stream s
$x : [1,m] \to X^\omega$	tuple of streams, with $x = (x_1, \ldots, x_m)$ for $m \in \mathbb{N}$
$x.n$	the tuple $(x_1.n, \ldots, x_m.n)$, if $n \in \mathbb{N}$
$\bar{x}$	time abstraction of timed stream x, a result from concatenating all sequences in x

$\mathit{Type}(c1\_out) = T1$ and $\mathit{Type}(c2\_in) = \mathit{Type}(c2\_out) = T2$. The component behavior is defined as

$$\overline{c1\_in} = \overline{c1\_out} \wedge \overline{c2\_in} = \overline{c2\_out}.$$

The *equational style* describes the behavior of a component as a set of equations that resemble functional language programs. Equations relate abstract states of the component and can be seen as state transition rules specified as equation expressions. The abstract state can refer to a condition of input and output messages, and can also encapsulate an internal state of the component [BS01].

Assumption/guarantee or assumption/commitment specifications provide logical expressions for the accepted (assumed) inputs of a component and the resulting (guaranteed) outcome in these cases. Thus they consist of two properties expressed in predicate logic.

Graphical specifications make use of tables and diagrams, which are more readable and intuitive for engineers, and directly translate to logical expressions. Tables are used in

FOCUS specifications as shorthand notations and structured representations for sets of formulas with a regular shape [BS01].

We describe higher level notations and specification techniques in subsequent sections.

3.1.4 Component Based System Model

A specification of components in FOCUS includes the description of structure and interaction behavior. Components interact via typed messages on channels. The external interface of a component has a structural (syntactic) and a behavioral (semantic) part. Interface behavior is described using streams based on the channel and message type definitions of the syntactic interface. Table 3.3 provides an overview of definitions for components. We follow the notational conventions used in [Bro05b]. In the following, we summarize definitions from [Bro05b, BKM07].

Table 3.3: Definitions for components

Notation	Informal Definition
C	set of channels
$I \subseteq C$	component input channels
$O \subseteq C$	component output channels
$Ch = I \cup O$	channels connected to the component
$Type : C \rightarrow TYPE$	data type of messages sent over a channel, with $TYPE$ being a carrier set of data elements
$(I \blacktriangleright O)$	syntactic interface
$\mathbb{H}(Ch) \stackrel{\text{def}}{=} x : Ch \rightarrow (\mathbb{N} \rightarrow M^*)$	channel history for a set of channels $Ch \subseteq C$, such that $x.c$ is a stream of type $Type(c)$ for each $c \in Ch$
$\vec{Ch} \stackrel{\text{def}}{=} \mathbb{H}(Ch)$	channel history for a set of channels Ch
$F : \vec{I} \rightarrow \mathcal{P}(\vec{O})$	semantic interface

The *syntactic interface* $(I \blacktriangleright O)$ of a component determines the channels Ch by which the component is connected to its environment, and the message types that are exchanged via the channels. The endpoints of channels are called ports. The input channels $I \subseteq Ch$ transport the messages from the environment to the component; the output channels $O \subseteq Ch$ transport the messages from the component to the environment.

The *semantic interface* of a component defines the interaction behavior and thereby the component's observable functionality. For a component F with the set of connected channels $Ch = I \cup O$, every channel $c_i \in Ch$ is associated with a stream set $M_{c_i}^{\omega}$, representing possible communication histories of messages $m \in M_i$ communicated over the channel c_i. $Type(c_i)$ yields the type of the channel c_i permitting only messages in M_{c_i}.

The interaction behavior of a component is expressed as relation between input stream sets and output stream sets. Nondeterminism is explicitly permitted by allowing more than one possible output stream set for an input. The component's behavior can be expressed through the following behavior function F, yielding a set of output stream sets.

$$F: \quad M_{I_1}^{\omega} \times \cdots \times M_{I_n}^{\omega} \rightarrow \mathcal{P}(M_{O_1}^{\omega} \times \cdots \times M_{O_m}^{\omega})$$

or in short

$$F : \vec{I} \rightarrow \mathcal{P}(\vec{O}) \ .$$

This function precisely defines the reaction to all possible input histories (input stream sets) on its input channels by returning the corresponding output histories (output stream sets) on its output channels. Figure 3.4 depicts a FOCUS component F with input channels $I_1, \ldots I_n$ and output channels $O_1, \ldots O_m$ and their respective types $M_{I_1}, \ldots M_{I_n}$ and $M_{O_1}, \ldots M_{O_m}$, where $F : \vec{I} \rightarrow \mathcal{P}(\vec{O})$.

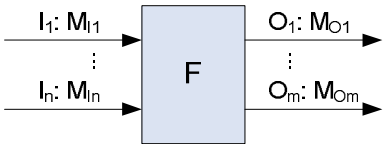

Figure 3.4: FOCUS component interface

Not all relations between input and output streams are permitted. FOCUS restricts the permitted specifications to those that fulfill the *strong causality property*, sometimes also called timing property because it axiomatizes the time flow. It ensures that a component's reaction to input messages can only be observed strictly after it received the respective input, guaranteeing causal component behavior. Such specifications are then called time-guarded and *strictly causal*. A mathematical consequence of this property, assuming $F.x \neq \emptyset$ for $x \in \vec{I}$, is that components yield a non-empty output for all input histories. For each input there is at least one output specified; we speak of *totally* defined component behavior. If a component defines exactly one output for each input, it is called *deterministic*. For details, see [BKM07].

Figure 3.5 shows an example of a simple component specification. It shows the "Transport" component from our running example with two input channels $c1\_in$, $c2\_in$ of data types $T1$, $T2$ and two output channels $c1\_out$, $c2\_out$ of the same types. A structural view on the component is depicted on the figure's right side. The left side shows the FOCUS specification, with an assertion expressing that each message on the $c1\_in$ input channel will eventually be produced on the $c1\_out$ output channel, preserving message order and respecting the timing property, but allowing arbitrary latency. Same, independently, for the $c2$ channels. This models a bidirectional reliable transport layer.

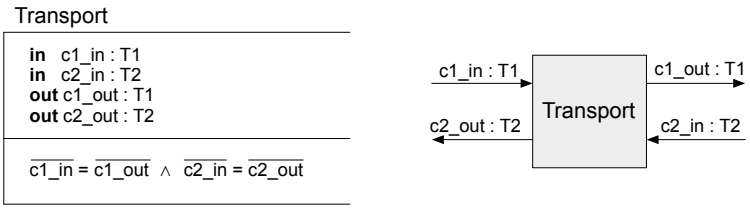

Figure 3.5: FOCUS specification example: "Transport" component

3.1.5 System Architectures Made of Components

Distributed systems and higher level components can be specified as a network of interconnected components, thereby providing a glass-box view showing their internal composition. In the following, we use the term system for such an interconnected set of components. The definitions in this section are summarized from [Bro05a, BKM07, Bro05b].

A system S consists of a set P of components and a set C of directed channels. Channels connect components that communicate with one another; they also connect components with the environment. Channel names are required to be unique within the system. Component identifiers $K \subseteq KN$ are assigned to components with the mapping $CName : KN \to P$. Channels have an associated type $Type : C \to TYPE$ constraining the transported messages. Table 3.4 summarizes these and further applicable definitions for systems, components and channels.

Each component $p \in P$ of the system has an associated focus behavior specification as introduced above

$$F_p \in Com[I_p, O_p] \ ,$$

where $Com[I_p, O_p]$ denotes the set of all input/output functions with given input channel set $I_p \in C$ and output channel set $O_p \in C$. Com denotes the set of all input/output-functions for arbitrary channel sets I_p and O_p. The function

$$\nu : K \to Com$$

maps component behaviors to component identifiers. $In(F_p)$ denotes the set of input channels of component p with behavior F_p and $Out(F_p)$ denotes the set of output channels.

A *system architecture* is a structural internal view on the system consisting of a network of interconnected components. Its nodes represent components and its arcs represent communication channels on which streams of messages are sent. A system architecture Ψ is denoted as (ν, O) for a system S with syntactic interface $(I \blacktriangleright O)$. Its definition is based on the mapping ν of component identifiers to behavior functions. Channels connect components according to their identifiers, requiring that the set of output channels for all components are pairwise disjoint. The set C contains all channels of the architecture. Output channels of the architecture to the environment are a subset $O = Out(\Psi)$ of all

output channels of the components that are not connected to another component[2]. The architecture determines which of these channels are made available to the environment, thus the denotation (ν, O). Internal channels of the architecture are direct links between two system components, determined as $L = C \setminus (I \cup O)$, cf. [BR07]. Table 3.4 summarizes the definitions for system architectures of components.

Table 3.4: System architecture definitions, based on [Krü00b, Bro05a]

Notation	Informal Definition
S, ENV	system and environment
P	set of system components
KN	universe of all component identifiers
K	set of identifiers $K \subseteq KN$ for components $p \in P$
$CName : KN \to P$	associates component identifiers
C	set of directed channels
CN	universe of all channel identifiers
$ch = (cn, cs, cd)$	triple of channel identifier, source and destination component of a channel $ch \in C$
$chn : C \to CN$	projects a channel $ch \in C$ on its channel identifier
$src : C \to (P \cup ENV)$	projects a channel on its source component or ENV
$dst : C \to (P \cup ENV)$	projects a channel on its destination component or ENV
$\tilde{C} \stackrel{\text{def}}{=} \{chn.c : c \in C\}$	set of channel identifiers $\tilde{C} \subseteq CN$ for channel set C
$I \subseteq C$	set of system input channels from the environment
$O \subseteq C$	set of system output channels to the environment
$L \subseteq C$	set of internal channels between components $p_1, p_2 \in P$
Com	set of input/output functions for arbitrary input and output channel sets
$Com[I, O]$	set of input/output functions for input channels $I \subseteq C$ and output channels $O \subseteq C$. $Com[I, O] \subseteq Com$
$In(F)$	set of input channels I for behavior $F \in Com[I, O]$
$Out(F)$	set of output channels O for behavior $F \in Com[I, O]$
$F_1 \otimes F_2$	composition of two components $F_1, F_2 \in Com$
$\nu : K \to Com$	associates component behavior
$\Psi \stackrel{\text{def}}{=} (\nu, O)$	system architecture

Figure 3.6, from [Bro05a], shows a graphical illustration of the "Transport" component of the system architecture discussed above, being constructed from a set of components $P =$

[2]Note that the definitions in this section require channels with one destination only. Other approaches such as [BR07, BKM07] permit output channels that are connected to another component and the environment simultaneously.

$\{Sender, Receiver, Medium1, Medium2\}$ and channels $C = \{x, c1, c2, c3, c4, y\}$. Subfigure (a) shows the components and their connected channels in isolation. Subfigure (b) shows how components are composed on shared internal channels $L = \{c1, c2, c3, c4\}$ in addition to the external system channels $I = \{x\}$ and $O = \{y\}$. Subfigure (c) shows the system architecture in a better readable form. The graphical representation is in form of a system structure diagram (SSD), a graphical notation for describing system architectures. The sets P and C directly follow from the graphical notation.

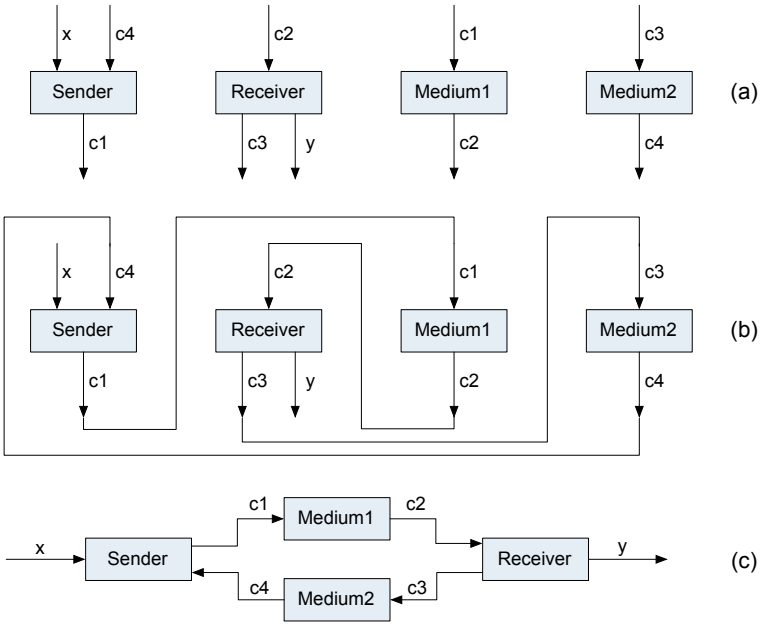

Figure 3.6: Graphical illustration of a set of components forming a system architecture, from [Bro05a]

3.2 A Formal Model of Services

In this section, we show how services can be represented formally, extending the formal model for distributed systems introduced above. As mentioned in the motivation, this definition is consistent with the general notion of a service as "a function of an application, described by an input-output interface, coordinating implementation level entities".

3.2.1 Service Formalization

A formal theory of services in distributed systems was introduced by Broy et al. [Bro03b] and extended in subsequent work [Bro05b, BKM07, Bro07b, GM09, Bro10]. Many defini-

tions presented in this section are taken from these references; please refer to the original sources for details. The theory provides a formal model of services and layered architectures. We present basic formal definitions in this section. Subsequently, we will provide methodology for structuring functional and logical architectures of systems using services.

Service Interface Services are formalizations of pieces of functionality that can coexist with other functionality for the same set of components. This leads to the definition of services as partial input/output functions. Services are a generalization of FOCUS components to partial interaction behavior relations on streams with a similar syntactic interface. Figure 3.7 depicts the syntactic interface ($I \blacktriangleright O$) of a service F. A service F is connected to a set of input channels $I = I_1 \cup \ldots \cup I_n$ and set of output channels $O = O_1 \cup \ldots \cup O_m$ with message types $M_{I_1}, \ldots M_{I_n}, M_{0_1}, \ldots M_{O_m}$ defined by each channel.

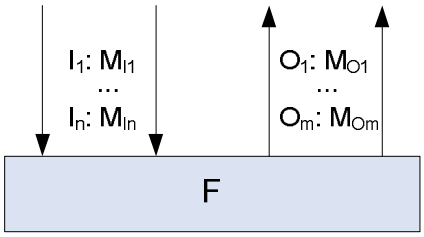

Figure 3.7: Service F syntactic interface

The service interaction behavior or *semantic service interface* is given by the function

$$F: \quad M_{I_1}^{\omega} \times \cdots \times M_{I_n}^{\omega} \rightarrow \mathcal{P}(M_{O_1}^{\omega} \times \cdots \times M_{O_m}^{\omega})$$

identical to the definition for FOCUS components; or in short

$$F : \vec{I} \rightarrow \mathcal{P}(\vec{O}) \ .$$

In contrast to components, which are defined for all inputs cf. Section 3.1.4, service behavior is only defined for a subset of all inputs. Thus component behavior is *total* and service behavior is *partial*. Components are the special case of totally defined services. The idea is to describe for a service under which conditions the functionality may be invoked and which effects this has. The service access conditions and effects define the service *protocol*.

Service Access Protocol Valid service input histories are elements of the *service domain*. The *service range* is the set of all service outputs produced for inputs from the service domain. These sets, *dom* and *ran*, respectively, are defined as

$$dom(F) = \{x : F.x \neq \emptyset\} \ , \ ran(F) = \{y \in F.x : x \in dom(F)\} \ .$$

The set of all service semantic interfaces with input channels I and output channels O is denoted by $\mathbb{IF}[I \blacktriangleright O]$. Service behavior functions must fulfill the *strong causality property* similar to component behavior functions; however only for inputs with non-empty output set [BKM07].

A service provides a partial view on the total behavior of a component. Through generalizing components in a theory of partial behavior specifications, services and service-oriented concepts can be defined. In addition, this theory enables us to separate specifications of system usage functions from logical architectures that realize the functions [BKM07, Bro10]. We will exploit the partial properties of service behavior definitions is later parts of this thesis. Table 3.5 summarizes definitions for services.

Table 3.5: Definitions for services

Notation	Informal Meaning
C	set of directed channels
$I \subseteq C$	set of service input channels
$O \subseteq C$	set of service output channels
$(I \blacktriangleright O)$	syntactic service interface
$F : \vec{I} \to \mathcal{P}(\vec{O})$	semantic service interface
$dom(F)$	service domain
$ran(F)$	service range
$\mathbb{IF}[I \blacktriangleright O]$	set of partial input/output functions for input channels $I \subseteq C$ and output channels $O \subseteq C$
$\mathbb{IF}$	set of partial input/output functions for arbitrary input and output channel sets

3.2.2 Formal Service Specification

In Section 3.1.3, we introduced several specification styles for FOCUS components. All of these translate to predicate logic expressions. In this section, we show how service interfaces can be specified based on partial input/output functions.

Services are pieces of functionality specified as partial functions. A specification style that emphasizes this partiality is the *assumption/guarantee style*. Here, a specification is split into two properties. The assumption property A defines the conditions on the domain that define valid inputs for the service. The guarantee property G defines the outputs for valid inputs. Such specifications characterize the service access protocol particularly well. In the following, we summarize the definitions of [BKM07].

An *assumption/guarantee specification* of a service behavior F with input history $x \in \vec{I}$ and output history $y \in \vec{O}$ is given in terms of *input assumption* A, with $A : \vec{I} \to \mathbb{B}$ and

output guarantee G, with $G : \vec{I} \times \vec{O} \to \mathbb{B}$. F is defined as follows:

$$A(x) = \langle \exists y \in \vec{O} :: y \in F.x \rangle \ ,$$
$$G(x, y) = \{ y \in F.x \} \ ,$$
$$F.x = \{ y : A(x) \wedge G(x, y) \} \ .$$

[BKM07] provides details and proofs. The service behavior specification F is a partial function. [BKM07] shows how a strictly causal and total component behavior F' providing the service F can be constructed through application of the *chaos closure*. The resulting behavior F' has the least number of outputs that satisfy the service behavior F.

Assumption/guarantee specifications are particularly suited to describe services because they highlight the service domain clearly by giving a service input assumption property. Nonetheless, all other basic FOCUS specification styles, namely relational, equational and graphical specifications can be applied similar to components. All of the specification styles generalize well to partial behavior functions for services. In the following section, we introduce another powerful specification style for services based on interaction patterns, graphically represented by Message Sequence Charts (MSCs).

3.3 System Development Methodology

In this section, we explain how the formalism and system model introduced above can be systematically applied to develop distributed systems, supporting the system engineering activities of requirements specification, system design, implementation, integration and verification. We use the term methodology for a comprehensive technique based on an underlying formalism with formal semantics, including description techniques for system specification and systematic model transformation operations.

3.3.1 Notations and Description Techniques

The behavior of a FOCUS component, defined as a function of input streams to sets of output streams, determines the component's observable interaction behavior. The behavior of a system of components results from the composition of multiple interconnected component behaviors, assuming a system-wide global discrete clock. System interface abstraction can be applied if only the black-box system interface is relevant.

In Section 3.1.3, we listed the four basic FOCUS specification styles that all result in component input/output functions. Section 3.2.2 generalized these specification styles for services and partial specifications. These specification styles remain very close to the mathematical formalism. Substantial research exists providing expressive, intuitive and scalable description techniques for various formalisms and system models. In the following, we introduce additional important description techniques based on the FOCUS formalism and its distributed system model.

State Machines State machine based description techniques are very popular because they enable a straightforward capture of regular component behavior in form of intuitive graphical specifications. State machines are also known as state automata and statecharts, and can be interpreted in various formalisms. State machines are very suitable to express component behavior as a succession of distinguishable states, with state changes triggered by interactions with the environment. State machines as description techniques can become complex if the number of component states is large or unbounded.

The state machine model described below [BKM07] is based on Moore automata [Moo56, HMU01], extended to infinite state spaces. State machines refer to the input and output channels of the component they specify. The data state of a component is described as a valuation of typed component attributes V. These attributes can represent local variables. The mapping

$$\eta : V \to \bigcup_{v \in V} \mathit{Type}(v)$$

provides a valuation of the component attributes with values out of $\mathit{Type}(v)$. The state machine describes a component's behavior by defining state transitions from one state to another. The space Σ of states is a partitioning of all attribute valuations. Often these discrete states are referred to as control state.

A state machine (Δ, Λ) with state space Σ and input and output channels according to the syntactic interface $(I \blacktriangleright O)$ is given by a set $\Lambda \subseteq \Sigma \times (O \to M^*)$ of initial states and initial output sequences, as well as a state transition function

$$\Delta : (\Sigma \times (I \to M^*)) \to \mathcal{P}(\Sigma \times (O \to M^*)) \ .$$

For each state $\sigma \in \Sigma$ and each valuation $u : I \to M^*$ of the input channels in I, by sequences, there is one successor state σ' and a valuation $s : O \to M^*$ of the output channels consisting of the sequences produced by every state transition, expressed as pair $(\sigma', s) \in \Delta(\sigma, u)$. Nondeterminism is possible by the definition of the state transitions as relation.

A state machine is called partial if the set $\Delta(\sigma, u)$ is the empty set for certain states σ and certain input sequences. By $SM[I \blacktriangleright O]$ we denote the set of all state machines with input channels I and output channels O. By SM we denote the set of all state machines, cf. [BKM07].

Interaction Based Notations This class of description techniques emphasizes the communication between components, represented in form of sequences and collaborations, sometimes also called interaction patterns. Various flavors of interaction based notations exist, including UML [OMG11b] collaborations with sequence and collaboration diagrams, Life Sequence Charts (LSCs) [DH01] and Message Sequence Charts (MSC) [IT96]. These notations are well suited to specify the interaction behavior of distributed systems, because they provide an intuitive representation of complex flows of information between

distributed components. Interaction based notations define sequences of messages exchanged between interacting entities, such as systems, components and objects. Each entity encapsulates its own state and solely exchanges information through messages. In this way, such entities resemble the properties of FOCUS components. These notations reach their limit when interaction complexity gets too high, for instance when the number of interacting components is large, interaction sequences vary significantly dependent on component state, and exceptional or forbidden behaviors need to be specified.

A particularly relevant description technique based on Message Sequence Charts (MSC) with full formal methodology is published in [Krü00b], building on the FOCUS formalism. We describe it's core definitions below, and compare it to related notations and semantic interpretations such as such as Life Sequence Charts (LSCs) [DH01].

3.3.2 Service Specification With Interaction Patterns

Interaction patterns can be used to specify system services; they realize partial specifications constraining the behavior of multiple interacting service components. Because each service provides only a partial view on system behavior, the full system behavior can be obtained by combining all service specifications, respectively their interaction patterns.

Krueger provides a formal semantics of interaction patterns and Message Sequence Charts (MSCs) [IT96] as graphical specification technique [Krü00b]. Extensions exist towards a comprehensive service-oriented development methodology. We leverage large parts of the MSC based interaction patterns and their formal interpretation in our service-oriented process; we describe where we extend the originally introduced methodology. In the following, we provide a brief introduction to this methodology, covering MSC specification techniques and their semantic interpretation.

Basic Definitions The semantic interpretation of interaction patterns defined in [Krü00b] is based on the formal definitions and the system model of FOCUS as introduced in Section 3.1, extended by definitions of the precise meaning of the introduced Message Sequence Chart dialect and the associated development methodology. This methodology [Krü00b] was developed independently of the theory of partial service functions as extensions of FOCUS presented in [Bro03b, Bro05b]. Both theories are closely related and have the same objective: to provide formal definitions and development approach for partial behavior specifications in distributed systems based on FOCUS. In the following, we introduce the semantics definitions for interaction patterns, summarizing definitions from [Krü00b]. We continue applying the notational conventions introduced above.

The formal interpretation of interaction patterns is built around a system of components P, connected by directed channels C, carrying messages of type M. Message parameters can be expressed by providing a further structuring of M. Each component $p \in P$ has a

set of states S_p. The state space of the system S is defined as

$$S \stackrel{\text{def}}{=} \Pi_{p \in P} S_p \ .$$

The set S_p of states of a component p can be structured explicitly by naming any local variables of p together with their types. A valuation of p's variables with values of the corresponding type yields a state of p. These definitions do not distinguish control and data state; the control state can be interpreted as partition of a component's state space. Components represent the structural entities of the system model, similar to the FOCUS system model. In contrast to the FOCUS model, components maintain a state and are not directly associated with an input/output behavior function.

Streams are used to define system behavior. Similar definitions to FOCUS apply. Every channel $c \in C$ is associated with the histories of messages on c in the order of their occurrence as an infinite stream of messages. The underlying model of time assumes a discrete global clock that drives the system, consistent with the timed stream model in FOCUS. Discrete time is represented in specifications by the set $\mathbb{N}$ of natural numbers. Components communicate asynchronously by sending and receiving messages on channels without blocking; in each time slot $t \in \mathbb{N}$, multiple messages can be sent or received. Specifications must respect the timing property thus enforcing strong causality, by producing output messages that only depend on input messages received in the previous time slot or earlier. The succession of system states over time is defined as an element of the set S^∞.

A Semantics of Interaction Patterns Using the above definitions, [Krü00b] defines the *semantics* of a system with channel set C, state space S, and message set M as

$$\mathcal{P}((\tilde{C} \times S)^\infty) \ .$$

Any element (φ_1, φ_2) of a system's semantics definition consists of a valuation of the system's channels ($\varphi_1 \in \tilde{C}^\infty$) and a description of the system state over time ($\varphi_2 \in S^\infty$). Nondeterminism applies if there exists more than one element in the semantics of a system. Table 3.6 summarizes the semantic definitions for modeling system behavior.

In the following, we briefly show some of the semantics definitions for interaction patterns from [Krü00b]. Note that these semantics do not cover the full official MSC standard but a significant subset. The semantics provided by [Krü00b] deviate slightly from the standards semantics for reasons of specification modularization.

The universe of all Message Sequence Charts (MSCs) is denoted by $\langle \text{MSC} \rangle$. An MSC α is a specification of system behavior and in particular the message exchange between a system's components. Thereby, it constrains the set of possible behaviors the system or its components can exhibit. The semantics of MSCs are defined denotationally using the definitions and system model introduced above, through a mapping to the semantics

Table 3.6: Core definitions and system model for MSC semantics, based on [Krü00b]

Notation	Informal Meaning
P, C	sets of components and channels of a system
S_p	state of component $p \in P$
S	system state ($S \stackrel{\text{def}}{=} \Pi_{p \in P} S_p$)
M	set of message identifiers
$\tilde{C}$	channel valuation at a particular time point ($\tilde{C} \stackrel{\text{def}}{=} C \to M^*$)
$\tilde{C}^\infty$	overall channel history
S^∞	state history
$(\tilde{C} \times S)^\infty$	combined channel and state history
$\mathcal{P}((\tilde{C} \times S)^\infty)$	semantics domain for system behaviors
$\langle \text{MSC} \rangle$	universe of all MSCs
$\langle \text{MGS} \rangle$	universe of all messages
$[\![\alpha]\!]_u$	semantics mapping as constaint on possible system behaviors imposed by MSC α, with elements $(\varphi, t) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty$
$MSCR$	associates names with MSCs, $MSCR \subseteq \langle \text{MSCNAME} \rangle \times \langle \text{MSC} \rangle$

domain of components, states, channels and streams. Each MSC translates to a set of channel and component state valuations for all channels and components in the system.

An MSC α constrains the possible system behaviors only for a certain time period. For every $\alpha \in \langle \text{MSC} \rangle$ and every $u \in \mathbb{N}_\infty$, exists a set

$$[\![\alpha]\!]_u \in \mathcal{P}((\tilde{C} \times S)^\infty \times \mathbb{N}_\infty) \ ,$$

where any element of $[\![\alpha]\!]_u$ is a pair of the form $(\varphi, t) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty$. For each such pair, φ describes an infinite system behavior and t the time when the behavior ends. Together with u, this forms the time interval $[u, t]$ indicating when the MSC behavior constraints are active. The MSC makes no statement about the system behavior outside of this time interval. MSC specifications do not contain absolute timing information. [Krü00b] proves that the behavior modeled by an MSC is independent of the time point at which the behavior starts; MSCs are independent of absolute time.

Note that MSCs do only indirectly constrain the behavior of the components that are part of the specifications. There is no explicit notion of assigning total input/output functions to component interfaces, such as with FOCUS components. Component behavior specifications are treated somewhat like the partial service behavior functions introduced above. We will explain this below when discussing methodology.

The relation *MSCR* associates names with MSCs; names are required to be unique within one specification. The function $msgs : \langle \text{MSC} \rangle \rightarrow \mathcal{P}(\langle \text{MSG} \rangle)$ determines the messages contained in an MSC. From the messages, we can infer the channels, because each message is defined as a triple of name, source and destination component.

The semantics mapping $[\![.]\!]_u$ defines an equivalence relation on MSCs. Two MSCs $\alpha, \beta \in \langle \text{MSC} \rangle$ are *semantically equivalent* with respect to time $u \in \mathbb{N}_\infty$, denoted by $\alpha \equiv_u \beta$, such that

$$\alpha \equiv_u \beta \quad \stackrel{\text{def}}{=} \quad [\![\alpha]\!]_u = [\![\beta]\!]_u \ .$$

The semantics of MSCs are defined by structural induction over the elements of the MSC notation. The following list contains the core elements and operators:

- *Empty MSC:* The neutral element for MSC specifications describing arbitrary system behavior that begins and ends at time u, defined as

$$[\![\mathbf{empty}]\!]_u \stackrel{\text{def}}{=} \{(\varphi, u) : \varphi \in (\tilde{C} \times S)^\infty\}$$

- *Arbitrary Interactions:* Any arbitrary system behavior starting at time u without upper time bound, defined as

$$[\![\mathbf{any}]\!]_u \stackrel{\text{def}}{=} \{(\varphi, t) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty : t \geq u\}$$

  any subsumes all possible behavior, i.e. for all $\alpha \in \langle \text{MSC} \rangle$: $[\![\alpha]\!]_u \subseteq [\![\mathbf{any}]\!]_u$

- *Single Message:* An MSC with only one message m on channel c, denoted by $ch \triangleright m$ and defined by

$$[\![ch \triangleright m]\!]_u \stackrel{\text{def}}{=} \{(\varphi, t) \in (\tilde{C} \times S)^\infty \times \mathbb{N} : t = \min\{v : v > u \wedge m \in \pi_1(\varphi).v.ch\}\}$$

  The specification requires the message to happen within finite amount of time.

- *Sequential Composition:* Denoted by $\alpha \,;\, \beta$, composes two MSCs α and β, such that both behaviors are active in sequential non-overlapping time intervals, and completed within finite amount of time. Note that the definitions here exclude weak sequential composition (such as defined by the MSC standard), where parts of the behavior can overlap as long as individual partial message orders are maintained.

- *Guarded MSC:* Denoted by $p_K : \alpha$. Constrains system behavior given by an MSC α only if the given precondition p_K holds at time $u \in \mathbb{N}$. After time u the states of the components may change such that the predicate $[\![p_K]\!] \in \mathcal{P}(S)$ no longer holds. The semantics is defined by

$$[\![p_K : \alpha]\!]_u \stackrel{\text{def}}{=} \{(\varphi, t) \in [\![\alpha]\!]_u : \pi_2(\varphi).u \in [\![p_K]\!]\}$$

- *Alternative:* Denoted by $\alpha \mid \beta$, defined as the union of the semantics of the two operand MSCs. The operands must be guarded MSCs and the disjunction of the guards must yield true. The semantics is defined by

$$[\![\alpha \mid \beta]\!]_u \stackrel{\text{def}}{=} [\![\alpha]\!]_u \cup [\![\beta]\!]_u$$

  Guard conditions are evaluated at time u, leading to a choice of exactly on subsequent behavior. If guard conditions are true for more than one MSC, nondeterministic choice is applied to select one MSC behavior.

- *Interleaving:* Denoted by $\alpha \sim \beta$, indicates the union of two MSC behaviors with causally unrelated interactions, i.e. components do not share the same messages. Also called parallel composition.

- *Join:* Denoted by $\alpha \otimes \beta$, the join operator composes two MSCs similar to their interleaving with the exception that the join identifies common messages, i.e. messages on the same channels with identical labels in both operands. Join composition is methodologically relevant when separating different functional concerns into separate specifications that should later be synchronized. [Krü00b] provides detailed discussions for the relevance of the join operator and the consistency requirements that must be met for two MSCs to be joined.

- *Loops:* Several classes of loop constructs that enable the repetition of MSC behavior α, where the decision for another repetition is determined by either a guard condition or a repetition counter; further alternatives include unbounded loops with finite repetitions and infinite loops. Guarded loops are denoted by $\alpha\uparrow_{<p>}$, bounded loops by $\alpha\uparrow_{<m,n>}$, unbounded loops by $\alpha\uparrow_{<*>}$ and infinite loops by $\alpha\uparrow_{<\text{true}>}$.

- *References:* Inclusion of the behavior of a referenced MSC as part of the specification of an MSC. Reference occurs by name. If no MSC with the specified name exists, any behavior is possible.

- *Preemption:* Denoted by $\alpha \stackrel{ch \triangleright m}{\rightarrow} \beta$ indicates that the behavior of the MSC α should be immediately interrupted and continued with the behavior of MSC β once the message $_{ch \triangleright m}$ occurs. Preemption is useful for specifying exception handling, for instance.

- *Preemptive Loop:* Special case of preemption, denoted by $\alpha\Uparrow_{ch \triangleright m}$, where the behavior of the MSC α is interrupted immediately upon occurrence of message $_{ch \triangleright m}$, succeeded directly by a restarted α.

- *Trigger Composition:* Denoted by $\alpha\Uparrow_{ch \triangleright m}$ indicates a temporal relationship between two MSCs α and β; whenever the behavior specified by MSC α has occurred, then the behavior specified by β is inevitable and must occur within finite amount of time. Trigger composition enables the easy specification of liveness properties.

Figure 3.8 provides a domain model specifying the operators and their composition to interaction patterns. This defines the MSC dialect that we employ for the specification of interactions of distributed system components. Interactions break down to sequences of send and receive events, composed using the operators listed above.

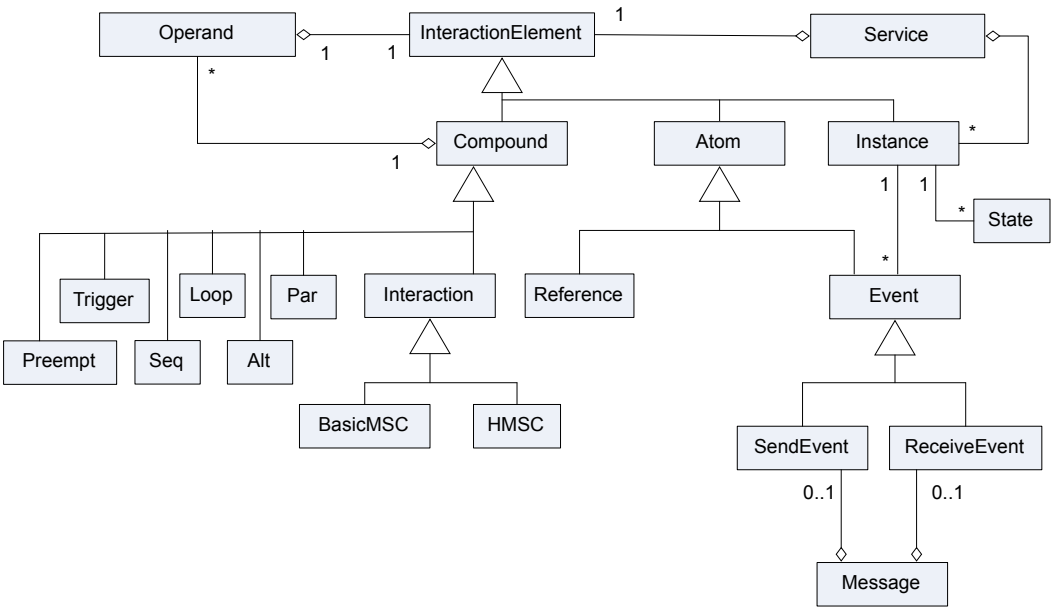

Figure 3.8: Domain model for Message Sequence Charts

Other MSC Semantics The work in [Bro05a] is based on the same FOCUS formalism but provides a slightly different interpretation of behavior in an MSC-based system specification. Instead of interpreting MSCs as partial orderings of messages that constrain the overall possible system behavior, it identifies causal sequences of send and receive message events for each of the components of an MSC. From these interaction sequences follow logical properties, which are guarded by a property capturing the state the interaction sequence assumes as precondition. Given all causal interaction sequences and their guards, the component behavior interface is determined by the conjunction of all its guarded interaction properties. The work furthermore defines a technique to structure and compose MSCs using state labels as additional means to specify guard conditions graphically, such that large system specifications can be decomposed in a modular way into smaller segments of interaction behavior, expressed by guarded MSCs. Algorithms can then detect completeness and consistency of such specifications.

Other MSC-based interaction specification techniques exist, such as Life Sequence Charts (LSCs) [DH01]. The authors focus on the specification of observable events in a distributed system that must occur. A system designer can designate select parts of the specification as always applicable; other parts may show possible behavior. The authors introduce the concept of "pre-charts" to enable the specification of universally applicable sequence charts based on a guard precondition. This enables system designers to specify conditions in the form "whenever something specific happens, something else must happen"—the "lifeness" properties of the system, cf. [DH01]. LSCs also allow the specification of "forbidden" behaviors. The authors outline a formal semantics based on state transition systems, extending the semantics provided for the MSC standard. At any instant in a system's

execution, each LSC has an activation state of "active", "terminated" or "aborted". Active charts constrain the subsequent possible message sequences depending on whether the chart was declared as universal, forbidden or existential.

Further MSC Semantics are discussed in [Krü00b], providing valuable insight into the details and intricacies of the original MSC standard, of scenario-based logical and graphical specifications techniques, and into related theoretical problems such as non-causal sequences, implied scenarios and automatic generation of component implementations.

Graphical Interaction Specifications MSC specifications can be constructed by combining interaction elements using the defined operators. The MSC standard provides a notation to depict such specifications graphically. Figure 3.9 shows a *Basic MSC* providing a specification for the "Compute Train Commands" functionality. The MSC specifies message exchange between three components, using the alternative and loop operators, with state labels indicating guard conditions for alternative choices and repetitions.

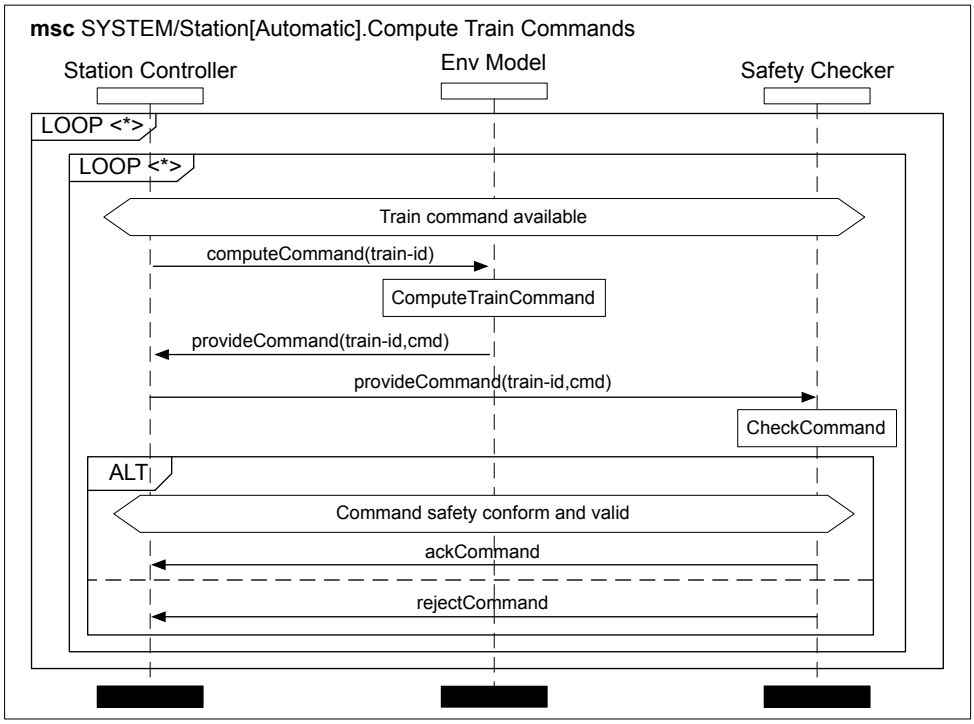

Figure 3.9: Graphical specification as basic MSC

High-level MSCs (HMSCs) are an additional graphical notation to structure sets of MSC specifications as flows. [Krü00b] defines HMCSs based on the introduced MSC semantics: HMCSs realize regular languages that can be effectively translated into MSCs. The translation algorithm makes use of automaton transformations, cf. [HMU01]. Figure 3.10 shows an HMSC from our running example that arranges three referenced MSC behaviors using

parallel composition. The behavior specified by the HMSC ends when all three parallel behaviors have ended.

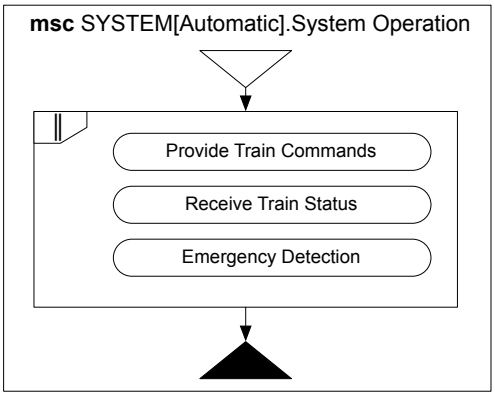

Figure 3.10: Graphical specification as High-level MSC

Methodology A service repository is a set of MSC specifications, including Basic MSCs and HMSCs. Each service is represented as an MSC. Services can be composed of other services, thereby establishing a service hierarchy. The root of the service repository—one MSC directly or indirectly referencing all other MSCs—represents the complete specification of the system behavior.

The vertical axes in MSC-based service specifications are called *roles* [Krü04] rather than components. This is to avoid confusion with the notion of FOCUS components introduced as total behavior functions. In the following section, we discuss methodology and some of the consequences of using roles instead of components in service specifications.

3.3.3 Specifying System Architectures

In this section, we show how system architectures can be composed and specified using the definitions introduced above. We summarize some definitions from [Bro05a, BR07, Bro10] applying the consistent notational conventions of this chapter.

Component Composition and Interface Abstraction The *interface abstraction* of a system S with system architecture (ν, O) and syntactic interface $(I \blacktriangleright O)$ yields the system black-box behavior as input/output function F. $F \in Com[I, O]$ is determined by the following formula with $x \in \vec{I}$, $y \in \vec{C}$ and channel set C:

$$F.x = \left\{ y|_O : y|_I = x \wedge \forall i \in K : y|_{Out(\nu(i))} \in \nu(i)(y|_{In(\nu(i))}) \right\} \ .$$

This formula applies function restriction of a function to a domain, essentially expressing that the output history of a system architecture is the restriction of a fixpoint for all the component behavior functions and their output channels, cf. [Bro05a]. In this way, the system is in itself a FOCUS component, which can be part of a larger system of components.

Component *composition* yields one component from two given components $F_1 \in Com[I_1, O_1]$ and $F_2 \in Com[I_2, O_2]$ by combining the syntactic interfaces and applying interface abstraction. Internal channels connecting both components will be hidden. The syntactic interfaces of both components must be *compatible*: components must have unique output channels and commonly used channels must have the same type.

The composition of two components with compatible syntactic interfaces is denoted by $F_1 \otimes F_2$, with combined channel set $L = L_1 \cup L_2$, input channel set I, output channel set O, and internal channel sets L_1 and L_2 constructed as indicated in Figure 3.11. See Figure 3.11 for formal definitions.

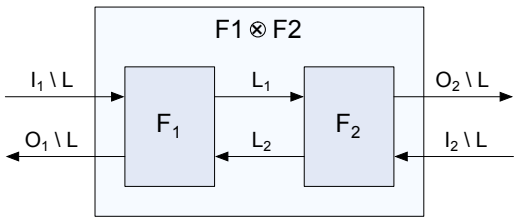

Figure 3.11: Component composition $F_1 \otimes F_2$, from [BR07]

The composition is a binary operator on two components. The resulting component can be composed again. Composition is commutative and associative; it is defined in a way such that the order of the operands does not matter and the result is uniquely determined.

Hierarchical Structuring of System Architectures A system architecture is a structural view on the components of a system and their interconnecting channels. In order to achieve increased scalability of specifications, additional systematic structuring spanning more than two levels (system and component) is required. System architectures constructed as component hierarchies provide this additional modularization.

Figure 3.2 above shows a system architecture, in which the system—the "Communication Network"—is composed of seven individual components. Channels connect the components with one another and the environment. The "Transport" part of the system shows a relatively isolated set of components realizing a modular functionality in this example: transporting information via a communication medium from a sender to a receiver. This makes it a good candidate to be modeled as a separate component that hides all details within. The architecture diagram in Figure 3.2 does not show it as a component; it has no ports, and channels are crossing right through its boundaries.

In the following, we show how hierarchy is defined in the FOCUS formalism and how it can be applied to construct system architectures systematically. We closely follow the definitions of [Bro07b].

A *hierarchy* for a finite set of nodes K of component identifiers is an acyclic directed graph (K, V) where $V \subseteq K \times K$ is a set of labeled arcs. A hierarchy is a partially ordered set. A hierarchy (K, V) is called *rooted hierarchy*, if there is a node $r \in K$ called the root, such there is a path from r to each other node. A rooted hierarchy is a partially ordered set with a least element. A rooted hierarchy is called a *tree*, if the path from the root to each node is unique. A tree is a hierarchy where nodes do not share successors. We call the successors of a tree node child nodes and the predecessor parent node. Nodes that have no successors are called leaf nodes.

Figure 3.12 shows two trees depicting the system architecture shown in Figure 3.2. Subfigure (a) shows a flat hierarchy with the "Communication Network" system node on the top. All components of the system are located on the same level below. Arcs indicate a hierarchical relationship as expressed by the relation V above. Note that there are two separate "Medium" nodes, indicating two instances of the same type. Subfigure (b) shows a tree with the same root. On the second level, however, are only four nodes. The "Transport" node has three child nodes; there are two instances of the node "Medium" in the system. Because of the abstraction provided by such a tree view, it is possible to hide multiple instances of the same component type with different channel connections. The hidden information needs to be present in other form, and potentially leads to different system implementations.

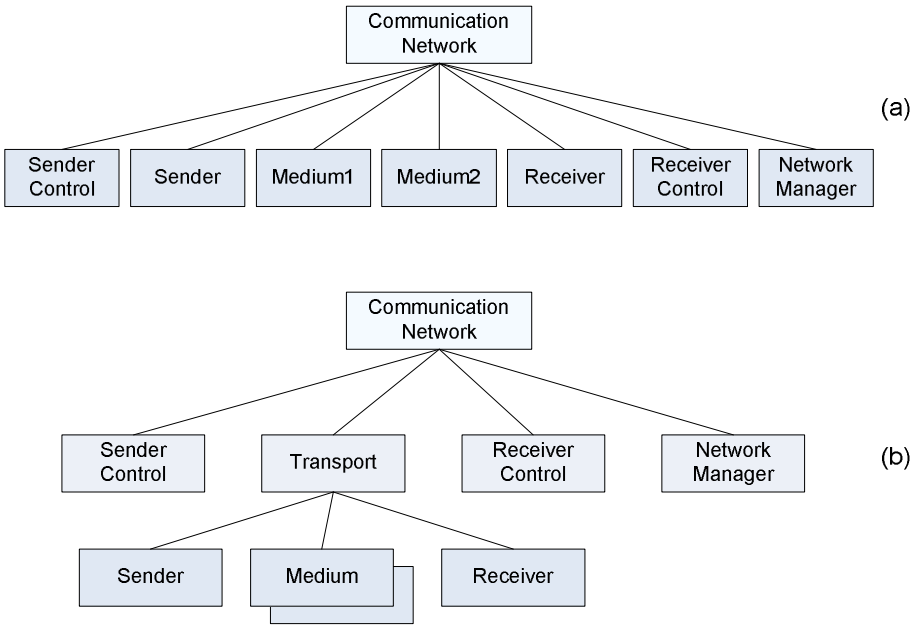

Figure 3.12: Graphical illustration of two possible component hierarchies

An *interpreted hierarchy* $((K, V), \varphi)$ is a hierarchy (K, V) with a mapping $\varphi : K \to Com$ that associates an interaction interface with each node of the hierarchy. An interpreted hierarchy is used both to define architectures and to define service hierarchies.

An interpreted tree $((K, V), \varphi)$ is called a *hierarchical (logical system) architecture*, if for each node $e \in K$ with a non-empty set $E = \{k \in K : (e, k) \in V\}$ of successors the composition $\otimes \varphi|_E$ is a refinement of $\varphi(e)$. In other words: the set of systems related to e forms a behavior that is a refinement of the interface behavior associated with node e.

[BR07] extends the definition of a system architecture (ν, O) to a *hierarchical system architecture* consistent with the definitions given above through induction. System architectures of deeper hierarchy level are constructed through building system architectures out of components and architectures of the next lower level[3]. A hierarchical system architecture forms an interpreted hierarchy, with mappings $\varphi : K \to Com$. The leaf nodes map to component interface behavior specifications and non-leaf nodes map to system architecture definitions also expressed as behavior interfaces, composing shallower system architectures and leaf components.

To identify the specific elements of hierarchical system architectures, we introduce the following notations. For a system S with component set P and channel set C, we identify by P_S all direct subcomponents of S on the level immediately below the system S itself. In general, by P_p, we identify the set of all direct subcomponents of any component $p \in P$. This is a shorthand notation for $sub.p \subseteq P$. We identify the parent component of a component $p \in P$ by $super.p \in P$, where $p \in P_{super.p}$.

3.3.4 Relating Architectures, Interfaces and Implementations

In order to support modularity and scalability of the design process, it is important to facilitate the translation between different specification techniques, in particular between component interfaces, component state machine implementations and system architectures. The following list includes transformations between representations, cf. [BR07]. In subsequent sections below, we show additional transformations relating service and component specifications of a system.

- A2I: System architecture to interface behavior
- M2I: Component state machine to interface behavior
- I2M: Interface behavior to component state machine implementation
- A2M: System architecture to state machine implementation

[3]The definitions in [BR07] explicitly distinguish interface behavior functions from state machine specifications. We see state machine specifications as one specification style for a component that can be expressed as behavior function.

Figure 3.13 depicts the transformations between specifications of the three different types. In the following, we briefly introduce the different transformations, with definitions from [BR07] using the notational conventions introduced in this chapter.

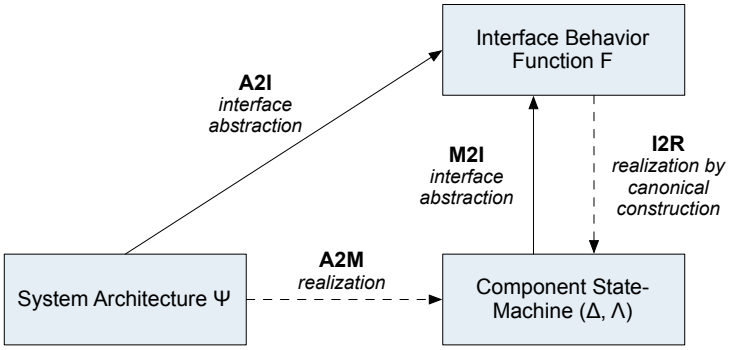

Figure 3.13: Transformation relations between architectures, interfaces and state machines

System Architecture to Component Interface Behavior Function (A2I) The mapping $A2I$ from a given system architecture $\Psi = (\nu, O)$ of components $p \in P$ with identifiers K to a component interface function F realizes an interface abstraction as defined in Section 3.3.3. It results from a composition of all components and the construction of the system's syntactic interface $(I \blacktriangleright O)$ with subsequent interface abstraction into a system behavior function F as constructed in the previous section.

The interface abstraction of the system architecture yields a composite component (interface), which can be part of another system architecture on a higher level. [BR07] shows that transformation is consistent with the composition operations used and realizes a morphisms of the form

$$A2I[\psi_1 \otimes \psi_2] = A2I[\psi_1] \otimes A2I[\psi_2] \ , \quad \text{for } \psi_1, \psi_2 \in \Psi \ .$$

State Machine to Interface Behavior (M2I) The mapping $S2I$ from state machines to interface functions realizes an interface abstraction. The syntactic interface $(I \blacktriangleright O)$ remains the same. The interface function can be derived from the state transition function of a state machine [BKM07]. Knowledge of internal component state is abstracted from and the resulting interface function

$$F_\Delta : \Sigma \times (O \to M^*) \to (\vec{I} \to \mathcal{P}(\vec{O}))$$

provides a black-box behavioral view for each state and initial output. The interface function is constructed by specifying relations for each input state and input pattern that relate input channel histories to valid output channel histories and outputs, for valid infinite sequences of states consistent with the state transition function. The construction ensures

that the resulting interface function is strictly causal. If the state machine is partially defined, the interface function is partial as well [BKM07].

Also for this transformation, [BR07] shows that it is consistent with the composition operations used and realizes a morphisms of the form

$$M2I[SM_1 \otimes SM_2] = M2I[SM_1] \otimes M2I[SM_2] \ , \quad \text{for } SM_1, SM_2 \in SM.$$

Interface to State Machine (I2M) An interface behavior function is the most abstract specification of component behavior providing a black-box view on the component. Deriving a state machine view does not yield a unique result and realizes an implementation step with many possible implementation variants. There is, however, a canonical construction of state machines given by [BR07], which is useful for verification and prototyping purposes.

[BR07] shows that *I2M* together with its inverse *M2I* realizes an isomorphism, such that

$$M2I[I2M[F]] = F \ , \quad \text{for } F \in Com.$$

System Architecture to State Machine (A2M) Deriving a state machine for a system architecture of components can be achieved through applying the canonical construction of state machines (I2M) for each of the interface behavior functions for the components of the system architecture, followed by a composition of state machines. Section 3.3.3 above defined composition for components specified as interface behavior functions, resulting in an interface function for the composite component together with a syntactic interface. The composition algorithm for state machines is fully compatible with the composition of interface functions [BR07].

The state machine for a given system architecture is constructed by applying state machine composition that entails a cross-product construction algorithm. This results in composite state machines with very large state spaces. The state space of the composite remains finite if all component state machines had a finite state space [BR07].

3.3.5 Relating Service and Component Specifications

It is very beneficial in terms of design methodology to be able to characterize the functionality of a system as a service architecture of independent and related services, as described above. This is particularly useful during the early phases of system design. Besides the functional decomposition, there may exist other factors of a more structural nature driving system design. These include structure of the target deployment environment, network topology, bandwidth, quality and latency, needs for replicating system elements to satisfy availability requirements and the existence of available component implementations.

Enabling both perspectives on system design and keeping them consistent requires formal relationships between both forms of specification. In the following, we discuss the relationship between a network of services and a system architecture of components, as well as transformations between both perspectives.

Services, Roles and System Architecture Services can be specified in form of MSCs describing interactions among a set of distributed entities. We assume a service hierarchy where there exists one root service specification for the system or a component, referencing all other MSC specifications. In Section 3.3.2, we showed a formal semantics of MSCs interpreting interactions among a set of components. [Krü04] adds semantics definitions to enable MSC specifications using roles instead of components, with subsequent mapping of these roles to components of the system architecture, using parametric MSCs. This greatly benefits development methodology. Roles introduce an additional level of abstraction into MSC-based specifications. Whenever a role occurs in an MSC service specification, its behavior is constrained through the input/output relations imposed by the specified interactions. Service specifications with roles can be authored somewhat independently from future target system architectures and can be reused for multiple different target architectures. Roles are the natural modular distributed entities to provide a system function and can subsequently be mapped to components as needed. Certain restrictions apply, for instance that the behavior of a role cannot be mapped to two distributed components.

In the following sections, we show how the behavior of a component "playing" one or multiple roles can be derived as input/output interface behavior function from an MSC specification. All role behaviors as implemented by such a component get composed, providing they have compatible syntactic interfaces and independent or consistent behavior definitions. We discuss a systematic development process employing services, MSCs, roles and component based system architectures in detail in the next chapters.

Transformations between Services and Components The following transformations relate service specifications and component specifications. These transformations are in addition to the four transformations between system architectures, interface functions and component state machines described above.

- S2I: Services to interface behavior
- S2A: Services to system architecture
- S2M: Services to state machines

Figure 3.14 shows all available transformations in the context of service-oriented development.

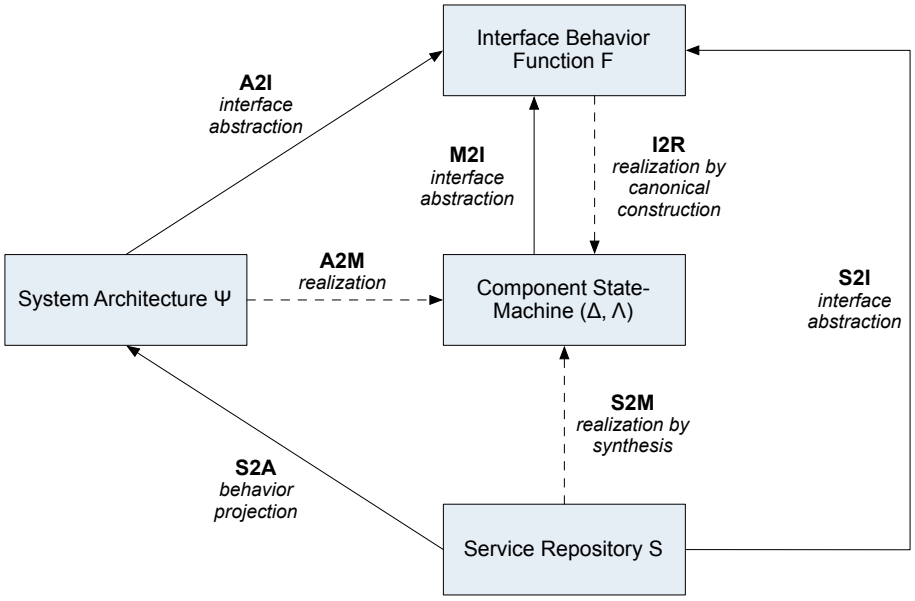

Figure 3.14: Transformation relations between services and components

Services to Interface Behavior (S2I) An MSC based service specification defines an interaction pattern among a set of roles. A mapping function exists between roles and components as explained above. A service specification thereby constrains the behavior of a set of system components. Semantics are defined through mappings to valuations of channels and component states. One MSC specification α constrains the behavior of multiple components $P_\alpha \in P$ connected through channels in $C_\alpha \in C$.

The component behavior projection

$$S2I(\alpha, p) : (\langle \text{MSC} \rangle \times P) \to \mathcal{P}(\vec{I} \times \mathcal{P}(\vec{O}))$$

yields a component behavior function F_p for a component $p \in P$ that is referenced by the MSC specification. The syntactic interface $(I_p \blacktriangleright O_p)$ of p is given by:

$$\begin{aligned} I_p &= \{(src.m, dst.m) \in C_\alpha : dst.m = p \wedge m \in msgs(\alpha)\} \text{ and} \\ O_p &= \{(src.m, dst.m) \in C_\alpha : src.m = p \wedge m \in msgs(\alpha)\} \ . \end{aligned}$$

The component behavior projection $S2I(\alpha, p)$ can be realized through the semantics construction described in [Bro05a]. This construction interprets MSCs as concurrent traces of interactions, constraining the referenced components. For one MSC and one component, a set of equations can be derived constraining component behavior. These equations are derived based on incoming and outgoing messages and assertions of component control state. Subsequent incoming or subsequent outgoing messages can be clustered into one equation. The set of all equations yields a predicate for a component. A number of MSCs referencing

the same component yields a number of such component predicates. These can be combined using logical conjunction, which requires that the behavior expressed by all MSCs results in deterministic component behavior. The component behavior function results from the logically combined component predicates after applying a *closed world assumption*. This means that the behavior of the component is exclusively specified by interaction traces given in the MSC specification, and all other behaviors are excluded [Bro05a].

Certain properties have to be true such that an MSC service specification can be projected into a component behavior function. For instance, the MSC specification must be causally consistent [Bro05a] and must yield deterministic component behaviors. If deterministic component behavior cannot be guaranteed, it may be possible to apply logical disjunction when combining component predicates in the above construction to yield nondeterministic component behavior with choice between possible applicable interaction sequences. This is useful to construct component behavior functions for early, not completely refined service specifications. Additional properties have to be true for non-lossy transformations, such that the composition of all component behavior functions results again in the specified MSC specification, for instance requiring causal MSCs [Krü00b].

The system interface behavior function F for an MSC service specification α can be constructed through composition of all component behavior interfaces with subsequent interface abstraction.

The service interface F is determined by the syntactic interface $(I \blacktriangleright O)$, constructed as

$$I = \{(src.m, dst.m) \in C_\alpha : src.m = ENV \forall m \in msgs(\alpha)\} \text{ and}$$
$$O = \{(src.m, dst.m) \in C_\alpha : dst.m = ENV \forall m \in msgs(\alpha)\} \ .$$

and the partial service behavior $F : \vec{I} \to \mathcal{P}(\vec{O})$, defined by

$$F = \bigotimes P, \text{ with } F.p = S2I(\alpha, p) \ \forall p \in P \ .$$

Services to System Architecture (S2A) The above described construction for component syntactic interfaces and behavior functions can be applied for all components referenced by the MSC specification in a service repository S, after applying the role-to-component mapping. The resulting set of channels and component specifications yields a system architecture for the service repository S.

Services to State Machines (S2M) The MSC based methodology of [Krü00b] provides a synthesis algorithm yielding executable implementations [Krü00b] in form of state machines. Starting point is a service repository S of MSC specifications, referencing a set of components P with communication channels C. An algorithm exists that can synthesize an optimized state machine implementation for one component $p \in P$ of the MSC service repository [KGSB99, Krü00b].

Figure 3.15 depicts the five steps of the algorithm. The algorithm takes an *MSC specification* as input in form of a set of MSCs and HMSCs. These MSCs need to be consistent and causal to yield reasonable results. Component state markers help to specify a seamless flow of interactions in particular for models of larger size. The set of MSCs is interpreted universally by applying a *closed world assumption* as described above. All MSC specifications together define the exact behavior of the system, eliminating any nondeterminism and partiality. We apply the algorithm for each component $p \in P$ in the specification, thus generating an equal number of state machines.

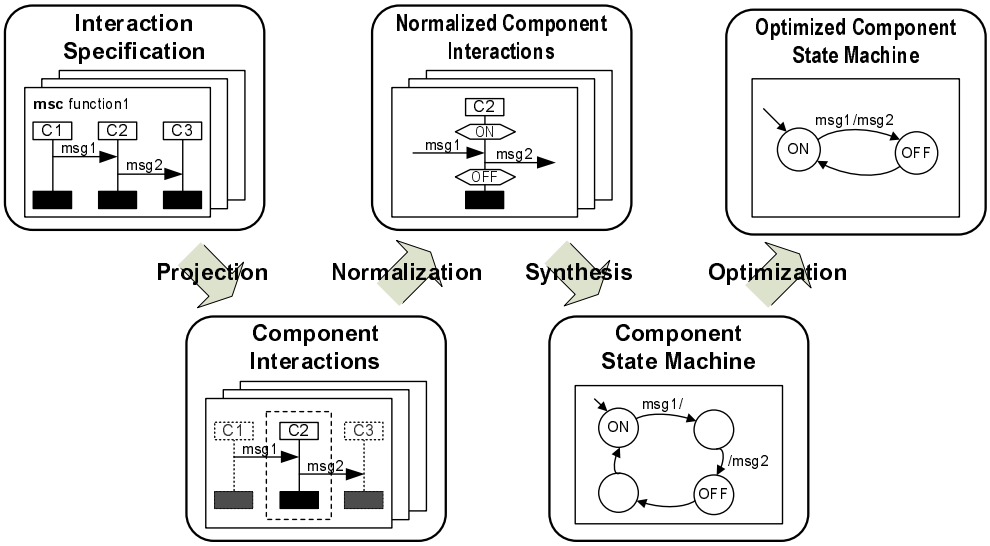

Figure 3.15: Component state machine synthesis algorithm

By *projection*, the algorithm eliminates all interactions that are not relevant for a component p and subsequently *normalizes* the resulting component specific interactions. Normalization adds missing start and end state markers, for instance. These interactions are input for component state machine *synthesis*. The algorithm translates each state label and guard appearing in any of the interactions into corresponding automaton states. It identifies the MSCs as transition paths and converts all message send/receive events into respective automation transitions between added intermediate states. The synthesis algorithm works fully automatic for *causal* MSCs and can handle choice, repetition, and concurrency/interleaving [Krü00b]. As final step it applies an *optimization* to the resulting state machine.

3.3.6 Refining Specifications

Developing a system specification is a complex task and requires a systematic and iterative approach. The same is true for developing an implementation that satisfies a given specification. In both cases, smaller and more abstract models need to be extended and

made more concrete during this process. Refinement is a way of relating two models or a model and its implementation, such that the refined model fulfills all or certain properties of the original model. An implementation can be seen as very detailed, directly executable refinement of the model.

FOCUS supports stepwise refinement and iterative development. It provides several types of *refinement*, each guaranteeing certain properties while the model is extended, design decisions are made and details are added. The following types of refinement are supported. We explain them in the sections below.

- Property refinement
- Interface refinement
- Conditional refinement

Property Refinement Sometimes also called behavioral refinement, property refinement is a fundamental refinement notion. It exists, when all logical properties of the refined specification imply the properties of the original specification. In an input/output relation, this is equivalent to reducing the number of outputs for the same inputs. The syntactic interface remains the same, see [BS01]. Through property refinement, it is possible to reduce nondeterminism of specifications by eliminating undesired output streams. It is also possible to impose timing constraints on previously untimed specifications.

In our formalism, property refinement can be defined as follows [BR07]. Let $F_1, F_2 : \vec{I} \to \mathcal{P}(\vec{O})$. Then F_2 is a property refinement of F_1, denoted by $F_2 \subseteq F_1$, if

$$\langle \forall x \in \vec{I} :: F_2(x) \subseteq F_1(x) \rangle \ .$$

Interface Refinement A generalization of behavioral refinement is interface refinement, which allows for the modification of the syntactic interface, such as the number of channels and types of messages. Through interface refinement, it is also possible to express components and systems on different levels of abstraction.

A specification F_2 is an interface refinement of specification F_1, denoted by $F_2 \subseteq_{(D,U)} F_1$, if two representation transformations D, U exist, with interfaces defined as

$$F_1 : \vec{I_1} \to \mathcal{P}(\vec{O_1}) \ , \ F_2 : \vec{I_2} \to \mathcal{P}(\vec{O_2}) \ , \ D : \vec{I_1} \to \mathcal{P}(\vec{I_2}) \ , \ U : \vec{O_2} \to \mathcal{P}(\vec{O_1})$$

such that $F_1 = D \circ F_2 \circ U$, applying function composition on set valued functions as defined above.

The downward transformation D makes an interface specification more concrete, while U raises it to a higher abstraction level. Figure 3.16 depicts the application of the transformations D, U to the input/output interface of F_1 to change the input/output representation.

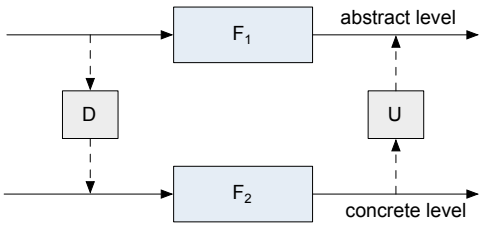

Figure 3.16: Interface refinement as commuting diagram, based on [BS01]

The downward transformation, D specified by a FOCUS interface transforms the syntactic interface and input streams as required, such that F_2 can refine the behavior of F_1 with the new interface. The upward transformation U makes syntactic interface and outputs compatible with the original component. Interface refinement is behavioral refinement, if both D and U transformations are the identity transformation.

Applications of interface refinement include changing the data type if messages and channels, modifying the communication infrastructure and extending the input domain [BS01].

Conditional Refinement Conditional refinement adds assumptions on the environment to both property and interface refinement. It is a generalization of interface refinement with one purpose of adding boundedness constraints to almost implementation ready specifications. Such boundedness constraints, for instance limits on internal buffer and queue sizes, often don't exist in earlier phases of development and would complicate initial specifications and formal reasoning on specifications [BS01]. Further applications include switching to implementable types, such as integers with fixed bit width, and imposing timing constraints on input [BS01].

Figure 3.17 provides an overview of the different refinement notions defined by the FOCUS methodology. Here, a fourth distinct class of refinement is depicted. Conditional property refinement is conditional refinement with unmodified input/output interfaces; as such it is a generalization of property refinement. For a detailed definition and discussion of refinement notions, see [BS01].

Glass-Box Refinement A specification of a system with (black-box) input/output interface and (glass-box) system architecture definition requires a special refinement notion, such that the constraints imposed by the system architecture remain maintained. Thus, glass-box refinement is a subset of the respective black-box property, interface and conditional refinements, maintaining the architecture constraints. The figure above indicates this through the outlined oblong.

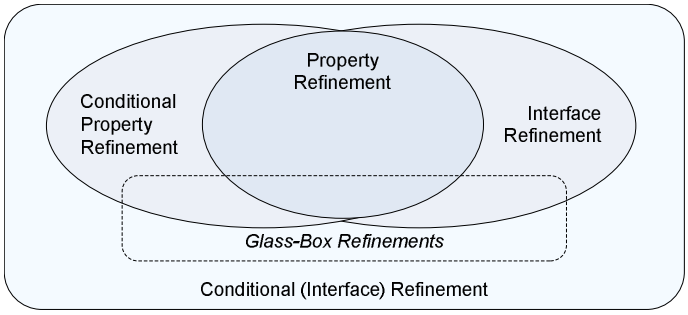

Figure 3.17: FOCUS specification refinement types, based on [BS01]

Service Refinement Service specifications are partial FOCUS behavior interface functions. A component is a special case of service with either an empty or total domain [Bro05a]. A service refinement is a generalized form of property refinement that allows the introduction of new channels. [Bro07b] defines service refinement as follows:

For partial service behavior interface functions $F_1 \in \mathbb{IF}[I_1 \blacktriangleright O_1]$ and $F_2 \in \mathbb{IF}[I_2 \blacktriangleright O_2]$, where $I_1 \subseteq I_2$ and $O_1 \subseteq O_2$, F_2 is a service refinement of F_1 if for all $x \in dom(F_1)$

$$\{y|O_1 : \exists x' : x = x'|I_1 \wedge y \in F_2.x'\} \subseteq F_1.x \ .$$

The service refinement is denoted as $F_1 \approx\!> F_2$. It assumes a behavior projection of the syntactic interface of F_1 to the syntactic interface of F_2. The respective input and output channel sets of I_1 and O_1 are in a *subtype* relationship to the channel sets I_2 and O_2, cf. [Bro07b]. Additional channels can exist in the refined service.

3.3.7 Service Development Methodology

In this section, we show how the previously introduced formal service notion and service specification techniques can be complemented by useful transformations and development methodology. Together with the already introduced formal model, these definitions are prerequisite for the systematic service-engineering development process with formal foundation that we will introduce in Chapter 5.

Subservice Relations In addition to service refinement, introduced above, further relations exist between two service interfaces to be leveraged, see [Bro07b, GM09]. A service F_2 is a *subservice* of a service F_1 if it is a service refinement and

$$dom(F_2) \subseteq \{x|I_2 : x \in dom(F_1)\} \ .$$

The service F_1 is called the superservice. The superservice provides the subservice and offers additional behavior. Hierarchies of services can be defined based on the subservice

relation. See [Bro07b] for additional details and a proof for the subservice relation as a partial order on all services.

[Bro10] introduces a *restricted subservice*, if, for services F_1 and F_2 as defined above with channel sets of F_2 in subtype relation to channel sets of F_1, both services are not in a subservice relation, but for a subset of input histories of F_2, $R \subseteq dom(F2)$ the subservice relation is true. This restriction eliminates unwanted input histories so that the subservice relation holds [Bro10].

Service Hierarchies [Bro10] introduces additional relations between services, such as service dependency and service hierarchies of various form. These definitions are similar to the definitions of hierarchies of components in system architectures, introduced in Section 3.3.3, but generalized to services and partial interface behavior functions.

Through the service refinement, subservice and hierarchy relations, it becomes possible to define a service architecture. A system's functional behavior can be decomposed into services, which can be broken down into smaller services down to a level of atomic functions. Service relations indicate the dependency of services within the service architecture.

Service Combination Two services can be combined, if there exists a superservice with both services as subservice [Bro10]. This operation combines two independent services into a functionally encompassing service. This enables the definition of a service architecture for a multifunctional system based on many existing independent services, for instance hierarchically. This works as long is there is no feature interaction [Zav01, Vog15] between the services, e.g. input channels that are shared among both services and input sequences that lead to nondeterministic or inconsistent output sequences. If feature interaction occurs, service composition may be an option. Alternative options are a revision of the services to prevent the feature interaction or the introduction of a coordinating component that coordinates input/output with the environment and manages both subservices independently.

Service Composition Service composition can be applied for services with mutual dependencies. Service composition is related to component composition introduced in Section 3.3.3. The syntactic interfaces of both services need to be refined first before service composition can occur. Service outputs that are inputs for the respective other service need to be separated into their own "internal" channels and connected as part of the composition operation. For additional details, see [Bro10].

Other Service Relations We have introduced *service slicing* [GM09] as a form of behavioral restriction of a service. It is a form of abstraction that derives a subinterface, preserving strong causality. Let F be a service interface on the channels (I, O), $I' \subseteq I$ a

subset of the input channels, $O' \subseteq O$ a subset of output channels be given. A function F' is called a slicing of F to the syntactic interface $(I \blacktriangleright O)$, with:

$$F'.x' = \{y | O' : \exists x \in I : x' = x | I' \wedge y \in F.x\} \ .$$

We also introduced *service projection* as a more general form of service slicing, allowing more flexibility with the service syntactic domain. In many cases, service slicing is closer to the way how humans intuitively experience the interaction with reactive systems, since Human-Machine-Interface (HMI) concepts are dominated by the idea of splitting the entire user interface into a channel-based structure [GM09].

In order to provide methodological guidance for how to decompose a service into subservices, we introduced the concept of an *Autonomous Service Partition*. This represents an independent piece of functionality, a subservice that is only influenced by messages on its input channels and not by other messages outside the syntactic interface of the autonomous service partition [GM09]. An autonomous service partition is called *minimal*, if it does not have a true autonomous service partition itself. The *Canonical Functional Decomposition* for a service F is the maximal set of services that each are minimal autonomous service partitions of F with the sets of input and output channels of each subservice pairwise disjoint. A canonical functional decomposition is called *perfect*, if F is completely covered by the subservices. A canonical decomposition is always unique. It may be the service F itself, if no true composition exists. Additional detail can be found in [GM09, Gru10].

3.4 Summary

In this chapter, we have introduced a comprehensive formal model of distributed systems. We have summarized definitions from the literature that are relevant as formal foundation for our service-oriented development process. All definitions in this chapter are based on consistent notational conventions and clearly indicate the original sources. Our formal model is based on the FOCUS theory and the concept of streams. The system model assumes a number of distributed components connected by channels. The formal model supports services as generalizations of components, specified by partial behavior functions. We have shown suitable description techniques for components and services, including a sophisticated methodology based on Message Sequence Charts. We have shown how system architectures can be defined based on networks of connected components, and how these architectures can be structured using multiple layers of a hierarchy. Transformations between various forms of system specifications, refinement notions and relations between services support a systematic iterative development of service-oriented specifications based on a consistent methodology. We will use the formal system model and accompanying formal methodology to derive and describe our service-oriented process and its artifact model in the subsequent chapters.

4 Service-Oriented Development Process Artifact Model

In this chapter, we introduce the artifact model for our service-oriented development process. It defines all system engineering work products and their dependencies. Central elements of the artifact model are the system architecture with system design models. First, we provide notational conventions and show how to structure a process artifact model. Then, we provide a comprehensive walk through all the artifact types of our process. Subsequently, we define the views that constitute the system design model and show how they can be represented as process artifacts. We define the process artifact model and design model views based on our formal system model and domain entity models. The artifacts were specifically designed to express the concepts of the formal model. We use our running example to illustrate artifacts, views, and notations. We list the benefits that our formal model provides as underpinning for the process artifact model.

Contents

4.1 Notations for the Process Artifact Model

This section explains the model specification techniques and notations used to define the artifact model for our development process.

Use of Domain Models *Domain modeling* techniques and notations are suitable for capturing concepts and knowledge within domains of interest systematically and abstractly, cf. [Eva03]. Domain models, in particular, are useful to capture a domain's concepts and their relationships graphically. These models are sometimes also called metamodels, knowledge models and concept models. We use domain models to capture the relationships between system design artifacts and design model elements. The domain of system design is very complex and requires the consideration of many aspects, including how to represent user observable system functionality, component behavior, component distribution, component interactions, data flow, system architecture, system quality properties, technical constraints, and infrastructure constraints. Providing descriptive domain models supports a clear articulation of the concepts and their relationships as applicable to our development process.

In the following, we introduce a comprehensive artifact model for our service-oriented development approach, specified by a set of domain models. Each domain model describes a subset of system design artifacts and their relationships. Multiple domain models may reference the same entities; this shows how the system design model views are integrated, for instance to express consistency rules between views and artifacts.

We use the UML2 *Class Diagram* notation [OMG11b] to represent domain models. Figure 4.1 provides a legend for the notation we apply. Boxes represent domain objects (entities, concepts, classes) and lines represent associations. Domain objects may provide additional behavior and attributes not indicated by the notation. Associations can be directed, depicted by an arrow, indicating a dependency from the source to the target entity. Bidirectional associations have no arrows. Directed associations can be specialized. Aggregation associations, depicted as line with an outlined diamond at the association source, denote an "has-a" relationship, or a "is-part-of" if read the other direction. Generalizations, depicted as a line with an outlined triangle at the relation end, denote an "is-a" relationship. This association type is often applied in the context of inheritance and subclassing, where the subclass entity inherits all the properties of the superclass entity and can specialize it further. Associations can have multiplicity values on either end and can be labeled. An arrow next to a label indicates the reading direction if the interpretation is unclear. Association entities exist to represent attributes for associations between two associated entities.

Stereotypes, indicated as a label in italics within double angular brackets in the upper right corner of an entity box, are generalizations of behavior; entity attributes are shared among all similarly stereotyped entities. In a way, a stereotype expresses a common superclass of

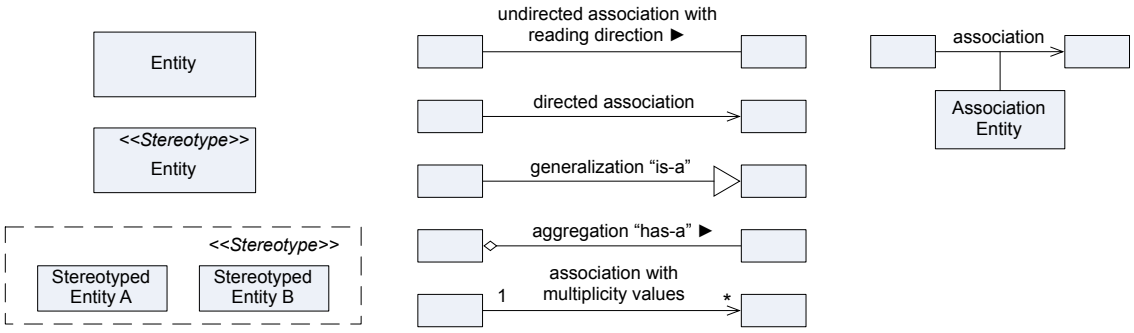

Figure 4.1: Domain model and class diagram notation legend

an entity, identified by the stereotype's name. In order to structure larger class diagrams and domain models more easily, we introduce some shorthand notations that go beyond the UML2 class diagram notation. A number of entities grouped together within a dashed rectangular area and a stereotype label in the upper right corner of the rectangle indicates applying the common stereotype to each entity within the rectangle. See [OMG11b] for details and semantics definitions for the UML2 class diagram notation.

4.2 Structuring the Artifact Model

In Figure 2.18, we characterized the constituent parts of a system development approach as foundation, concepts, notation, methodology, process and tools. We described dependencies between these parts. A process is comprised of artifacts, activities and other process elements. The artifacts of a process can be specified within an artifact model. In this section, we characterize the system development artifacts according to their purpose in the development process. Two structuring principles apply: by system engineering activity and by abstraction level. In the following, we introduce these principles and their significance for system development.

4.2.1 Structuring Artifacts by System Engineering Activity

There exists a natural ordering of the system engineering activities described in Section 2.2 with respect to the work products they require and produce, from requirements engineering to system design, implementation, integration and operations. Each activity takes information contained within a set of input artifacts and produces new information contained within output artifacts. An activity may be scoped, e.g. to the level of a specific system component. Performing activities adds information to the system development artifacts, sometimes increasing the level of detail of existing information.

This natural ordering of system engineering activities does not preclude that they can overlap, be skipped or performed out of order in development projects. Additionally, as we have shown in Section 2.4, it is often more practical to break the system down into smaller components and apply the system engineering activities somewhat independently on a component level, or to structure the development into increments and cycle through the activities multiple times.

Development processes define artifacts—often realized as structured documents—that contain system design information produced by respective system engineering activities. As shown in Section 2.5, using the example of the V-Modell XT, there typically exist specification, design, implementation, integration and verification artifacts. These artifacts may be related by product consistency rules, cf. Figure 2.17, or by some weaker form of referential consistency. From a process designer's point of view, it is suggestive to separate system development work products into discrete artifacts by engineering activity. One of the downsides of such a partitioning of the artifact model is that related information across development activities gets split up somewhat arbitrarily into the various development artifacts. Strong tool support and well-defined consistency rules can help keeping the various system development artifacts consistent and easy to navigate by the development engineer.

When working with an integrated system design model, one challenge is to partition the model's various views into discrete artifacts, potentially subject to independent authoring and review cycles. Our approach addresses this point by providing clear instructions for how to relate an integrated system design model and a discrete development process artifact model.

4.2.2 Structuring Artifacts by Abstraction Level

Whether system development artifacts get created in a "waterfall" approach or get refined iteratively, there is often a need to view consistent cross-sections across these artifacts that pertain to specific development activities or target audiences. For instance, being able to view all component specifications consistently across various system elements. "Abstraction levels" help to address these needs by defining layers of understanding of the system's design that hide complexity and detail of other layers from the current view. These layers are particularly beneficial for model-based development with its complex design models.

Abstraction levels provide an alternative structuring principle for development artifacts. When applied to system development processes, abstraction levels group model views and model operations with similar level of detail. Wild et al. [WFH+06], for instance, identify four abstraction levels that are relevant in the design phase of embedded automotive software systems, depicted in Figure 4.2. On the *service level*, system behavior is described as independent partial pieces of functionality. The *functional level* provides a totalized view on the entire observable system functionality. Distribution and partitioning of functionality into components are considered on the *logical cluster level*, whereas the *platform*

level describes technical details, such as the execution environment, the communication infrastructure and the actual deployment of the system.

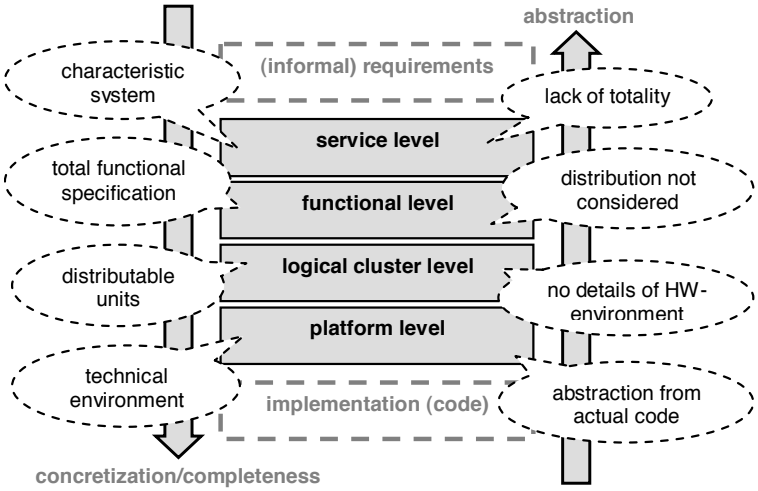

Figure 4.2: Abstraction levels for automotive software development, from [WFH+06]

The methodical use of abstraction levels in system development approaches varies. The approach of [WFH+06] uses abstraction levels to identify distinct intermediate work products. All products of one abstraction level need to be complete and consistent before products of the next lower abstraction level can be derived on that basis. This cycle can be repeated in order to realize multiple increments of the system. However, this might cause additional completeness and consistency checks for the products of the individual abstraction levels and potentially lead to substantial rework in case changes affect more than localized components.

4.2.3 Abstraction Levels for Service-Oriented Development

Our process artifact model applies both structuring principles: We define work products primarily associated with system engineering activities. This aligns with development processes such as the V-Modell XT. At the same time, we enable the engineers to work with a comprehensive system design model. We structure design model views along defined abstraction levels and define how they integrate into the process work products.

We interpret abstraction levels as focused, consistent views on the overall set of artifacts in a seamless development approach. This enables us to provide the view for an abstraction level anytime, for purposes of simplification and abstraction from details, without the need to manage these views as distinct artifacts of the artifact model. Model views remain consistent, because they are part of an integrated design model. In the subsequent sections, we describe our development process artifacts by system engineering activity, referencing

the respective abstraction levels for service-oriented development, depicted in Figure 4.3.

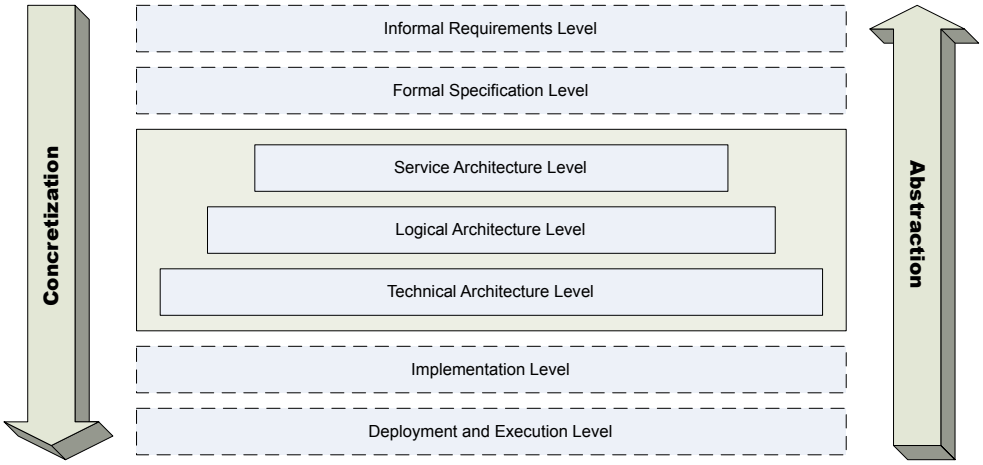

Figure 4.3: Abstraction levels for service-oriented development

Abstraction levels include:

- *Informal Requirements Level:* Comprises informal requirements that express goals of diverse stakeholders; goals may be conflicting and represent trade-offs. Developed as part of requirements engineering.
- *Formal Specification Level:* If existing, derived from requirements through thorough analysis and formalization, resulting in a consistent and comprehensive functional specification of the system.
- *Service Architecture Level:* Describes a service-oriented architecture of individual interconnected services together with a set of named interacting entities within the system and the environment.
- *Logical Architecture Level:* Defines a component architecture, comprising a functionally comprehensive design of the system structure and behavior, specifying interconnected, potentially distributed components. Design specifications remain on a logical level.
- *Technical Architecture Level:* Provides a technical design specialized to a deployment architecture, incorporating technical detail such as network topology, component mappings to specific middleware systems and implementation technologies.
- *Implementation Level:* Represents the results of system development and integration activities. Comprises individual and integrated realizations of the design specifications, and existing components.

- *Deployment and Execution Level:* Represents the deployed system in its target environment as well as artifacts related to configuring, installing and deploying the realized system.

For our approach, we particularly focus on the three abstraction levels related to system design. We assume the presence of some form of structured requirements or a functional specification. Artifacts on the *Service Architecture Level* define the behavior of the system as services offered by interacting system entities. Services are independent pieces of functionality. They can be combined with other services and mapped onto a component architecture. An essential purpose of this level is to enable iterative development of the functionality of the system together with a structuring of interacting system entities providing the functionality. We put special emphasis on providing models that enable partial behavior specifications, nondeterminism and refinement. The artifacts on this level specify the complete interaction behavior of the system, spread across multiple service definitions, without predetermining the actual system architecture of components.

Artifacts on the *Logical Architecture Level* group system entities from the service architecture level into well-defined components that can be implemented and distributed. Multiple services get combined into component behavior descriptions, removing partiality from specifications as well any remaining underspecification and nondeterminism. In case services cannot be combined into a component architecture meeting the requirements, an controlled revision of the service architecture needs to occur.

The *Technical Architecture Level* comprises artifacts that provide additional details regarding the instantiation of the components and their distribution within a concrete deployment. We consider this level the lowest level comprising system design artifacts. Subsequent activities cover implementation and integration, which are out of scope of the work described in this thesis.

4.2.4 Structuring the Service-Oriented Artifact Model

Our system development artifact model is structured into the high-level artifact classes shown in Figure 4.4. These classes are aligned with the abstraction levels for service-oriented development introduced above.

We base all development artifacts and design model views on a comprehensive metamodel, introduced in this chapter. This enables us to maintain them in a continuously integrated state. We support the embedding of design model views into process artifacts, which can be verified and published. All elements of the depicted domain model share the stereotype *System Development Artifact*, indicating that they belong to the development artifact model. This stereotype applies common properties defined in the artifact metamodel to all development artifacts, for instance relationships to responsible and authoring role assignments, and the applicability of product consistency rules. There exist no constraints on the form of the artifact at this level, such as document or specification model.

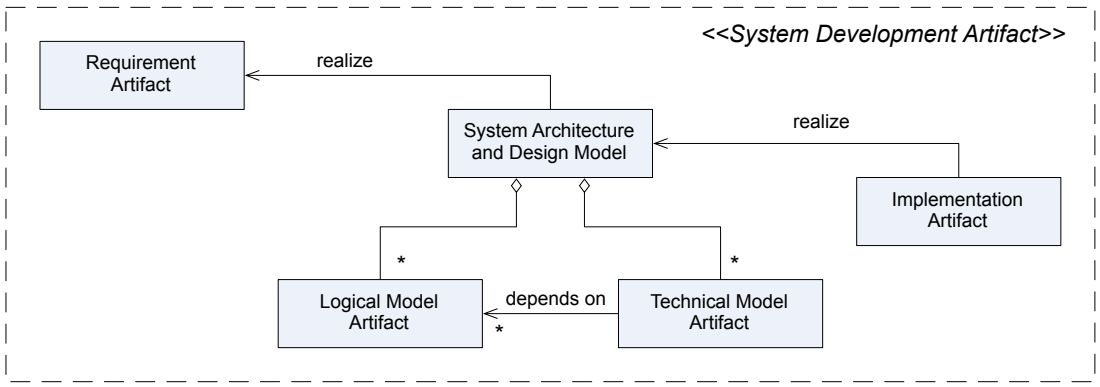

Figure 4.4: High-level system development artifacts domain model

We identify four groups of system development artifacts: Instances of *Requirement Artifact* contain informally and formally represented requirements, such as use cases, technical and project constraints, nonfunctional requirements, formal specifications and ancillary results of requirements engineering activities.

Central to our development process is an integrated *System Architecture and Design Model*. It realizes a primary system development artifact of a seamless system development process; it grows in size and in detail with each iteration of the main system development activities, and remains consistent for all revisions of system design. A *Logical Model Artifact* represents a "logical" part of the design model. These artifacts document the structural decomposition of the system into distributed components, and specify component interfaces and behavior. Analogously, a *Technical Model Artifact* represents a "technical" part of the design model; these artifacts document technology choices, communication infrastructure dependencies, middleware deployments, network distributions, replication and further artifacts of deployment configuration design activities.

Instances of *Implementation Artifact* include all direct results of software and systems implementation activities, such as source code, executables, deployable packages, software documentation etc.

The domain model in Figure 4.4 also identifies the high-level dependencies between these four artifact classes. These dependencies are characterized by two applications of a "specification–realization" pattern. First, the requirements artifacts provide the specification that the system architecture and design model realizes. Second, the system architecture and design model provides the specification for the actual implementation, with implementation artifacts realizing parts of the design model. This pattern is very important for scalable, modular system development; we will see it applied to other parts of the artifact model.

The two partitions of the system architecture and design model express a clear dependency. The logical model elements describe the system on a logical level, abstracting from technical details, specifics of the deployment infrastructure and the specific target deployment

configuration. The technical model elements contribute all these specifics to the comprehensive integrated model, by explicitly referring to elements of the logical model. This separation enables us to develop one compact and complete representation of the system design that can subsequently be specialized for different target environments. In case of changes to the system's deployment environment or technology set, the logical model ideally remains unchanged. The technical model needs to be modified—explicit references to the logical model elements help to make this modification as modular as possible.

The artifact model provides sufficient precision because of its well-defined metamodel and artifact definitions. It also provides flexibility for extension because of its modular definition of concepts and their dependencies. Our process can be seamlessly extended with additional process elements as needed—for instance to support domain specific concepts.

4.3 Service-Oriented Development Artifacts

In this section, we introduce all artifacts and their dependencies grouped by system development activity. This includes requirements, system architecture and design model, and implementation artifacts. We provide detailed explanations for all artifacts within these classes. We use domain models to relate the artifact model elements.

4.3.1 Requirements Engineering Artifacts

The activity of requirements engineering [SK98] provides an elicitation of requirements for the system under development. This is performed by system engineers in collaboration with domain experts and other stakeholders. In general, numerous stakeholders contribute requirements of different granularity and precision. Requirements are often expressed in the language of the application and problem domain; they may be incomplete, redundant, inconsistent and imprecise at first. All identified requirements need to be structured systematically, and written in a way to be testable, understandable and expressing one idea only. The activity of requirements analysis aims at distilling a complete, concise, consistent, redundancy free, and precise specification of the system out of the collected informal use cases, feature requests, nonfunctional requirements and given constraints. Figure 4.5 depicts the system development artifacts typically related to requirements engineering.

These artifacts include descriptive materials illustrating the context of the system, prescriptive materials framing frame system development and imposing conditions, a list of structured requirements and potentially a system specification that precisely defines system properties.

Descriptive materials include background materials, commonly in natural language text, describing the technical, commercial and scientific motivation of the system, the place

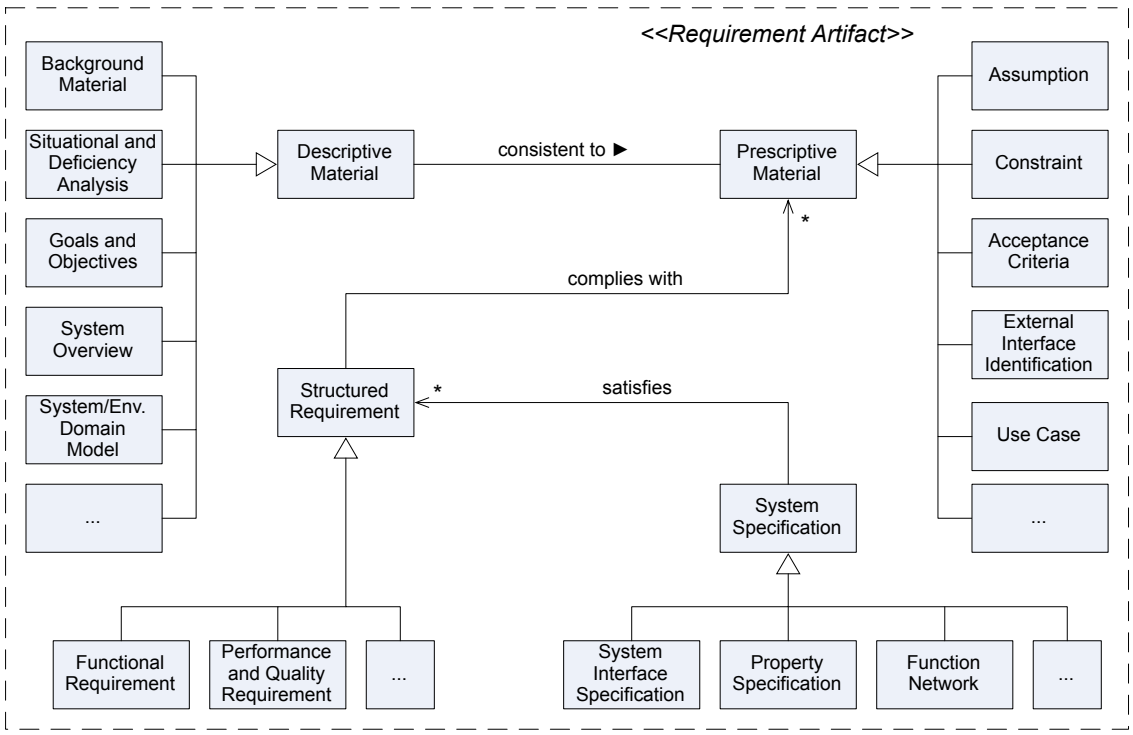

Figure 4.5: Requirements engineering artifacts domain model

it assumes within the environment, any stakeholders involved and context within an organization. A situational and deficiency analysis may exist, produced by a gap analysis activity, outlining potential benefits to be expected from the system. Goals and objectives for the system include statements that help making decisions where trade-offs are required. They are not binding, may be conflicting and express the high-level intent of stakeholders and users of the system. A system overview provides a high-level description of the system within its prospective environment and key system properties, for instance, a high-level concept graphic or the result of a conceptual design phase. Domain models capture knowledge about the system and its environment, help to increase language tangibility and provide a vocabulary for reference in requirements and system design.

Prescriptive materials express constraints and conditions applicable to the development of the system and its operational form, often rigorously written and carefully reviewed and approved. This includes assumptions on the system's situational context and environment, to be treated as assertions. A system design may require adaptation should these assumptions change over time. Constraints may identify applicable industry standards compliance and a development project time line. Acceptance criteria, provided by the stakeholders of the system and often included as part of a contract, explicitly state how the system acceptance process will be performed. The external interface identification provides a list of system interfaces with other systems and to users. Use cases provide prescriptive, ideally testable scenarios of exemplar system use by representative users. In some system

development approaches, use cases replace explicit lists of requirements.

Structured requirements are a primary result of the requirements engineering activity. Typically formulated in natural language text, such requirements undergo prioritization, cost-benefit-analysis, selection and review steps. Only requirements selected as the result of requirements analysis[1] are binding for the design and implementation phases of the system. Structured requirements comprise functional requirements defining the behavior of the system, and performance and quality requirements[2] describing the operational properties of the system. These are often referring to the system's "ilities", such as such as usability, scalability, availability, maintainability, extensibility and security. Other requirements may exist, such as interface requirements. Requirements may be structured hierarchically, for instance into user, system and subsystem levels. Requirements can be "allocated" to lower level requirements, which in turn may serve as "satisfiers" of higher level requirements. For instance, satisfaction of all allocated subsystem requirements for a system requirement may be necessary for that requirement to be satisfied.

A *System Specification* is an expression of essential requirements within a formal or semi-formal language with enough expressiveness and defined semantics to state the core structural and functional properties required of the system. Many model-based development approaches aim to formalize and structure requirements as much as possible to narrow the gap to subsequent system development activities, and to also enable formal verification of system designs and implementations against the specification.

There are several important dependencies between requirements engineering artifacts. Structured requirements must comply with prescriptive materials. To the degree applicable, descriptive materials must be consistent with prescriptive materials. A system specification must satisfy the list of structured requirements. Because of the informality of the information in requirements engineering, much of the analysis and enforcement of artifact consistency rests on human analysis and on effective tool support. Some requirements management tools provide tracing of individual structured requirements to information sources, cf. [Aut06a, MRS06, SFGP05].

4.3.2 Logical Architecture and Design Model Artifacts

The activities related to system architecture design are commonly carried out by system and software architects with significant technical and domain knowledge. During these activities, frequent interaction with prospective system users, stakeholders, requirements analysts and system implementers is required. The main artifact related to these activities is the system architecture and design model. In this section, we explain the logical part of this model; the technical part will follow in the next section.

[1] also called requirements controlling
[2] often called nonfunctional requirements

Logical Architecture and Design Model Views Figure 4.6 provides a domain model depicting the artifacts of the logical architecture and design model for our development process. The design model artifacts are represented by entities of class *Logical Model Artifact*. The actual described system elements, such as subsystems, components and classes are represented by instances of the class *Logical Model Element*.

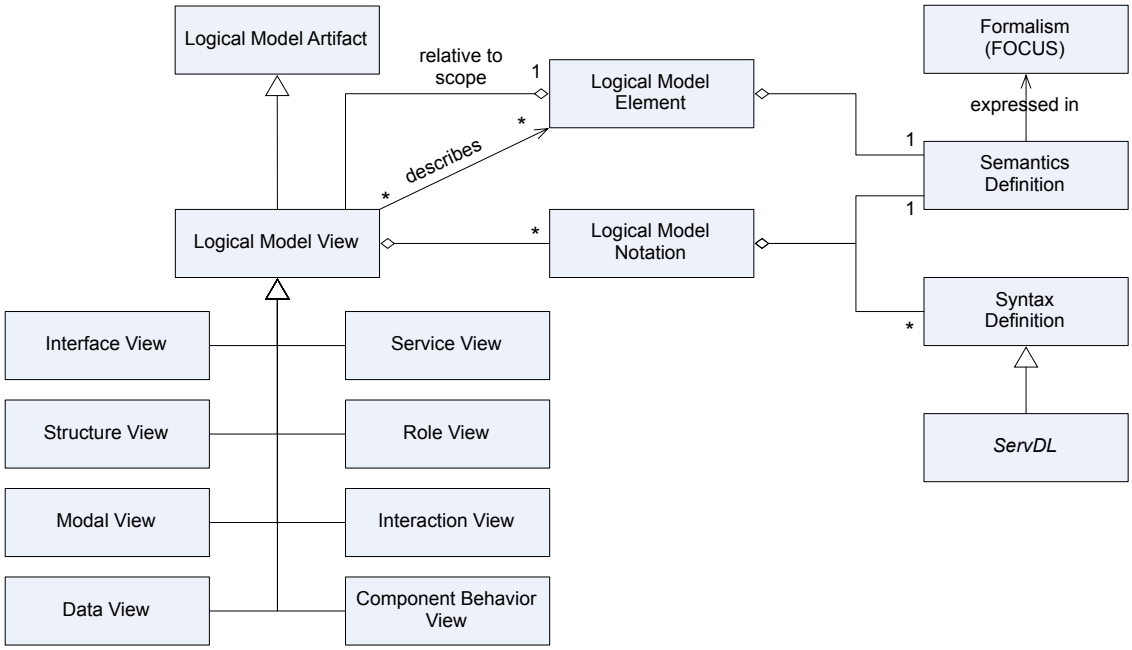

Figure 4.6: Logical model views and notations domain model

The integrated model offers a number of *Logical Model Views* that provide purpose specific representations of the model. These views satisfy the views required by IEEE-1471 [Arc00] for architectural descriptions. Each such view is associated with suitable *Logical Model Notations*. Model views are scoped to elements of the hierarchical system architecture. For instance, a view can show the entire system, a component or a subcomponent only. Each model view—in practice often represented by one diagram—targets a number of model elements. The following lists all types of model views for logical architecture model elements and the system itself:

Interface View Provides interface specifications for a component. Interfaces provide black-box views on a component; they provide essential guarantees when composing components.

Service View Identifies a component's services as first-class entities. Services are characterized by syntactic and behavioral interaction interfaces constraining the component and its environment.

Structure View Shows the decomposition of a component into subcomponents. Shows the communication links between subcomponents and with the environment.

Data View Specifies a component's data state and applicable data types. Specifies information exchange between components as data flows.

Modal View Shows a component's operational modes and valid mode transitions. The underlying premise is that exactly one mode is active at any given time; mode transitions result in change of component behavior.

Role View Specifies a component's roles within interactions. These roles can be mapped to any combination of subcomponents and component modes. Roles are an abstraction and indirection that avoid the need for directly referencing logical components in interaction definitions.

Interaction View Specifies the exchange of information between roles through discrete messages. Interactions specify service behavior in a partial way that does not constrain role behavior outside of the interaction.

Component Behavior View Specifies the externally observable behavior of a component through its interfaces. This specification is total and defines the observable behavior for all input conditions.

Each view represents a number of model elements using notations applicable to the view type. Each notation has one or more *Syntax Definitions*, such as textual and graphical syntaxes. Model elements and notations have unique *Semantics Definitions* in our formalism *(FOCUS)*. We will provide detailed definitions for views and notations in Section 4.4 below. We also introduce the *ServDL* textual syntax for central notations of our architecture model.

View Dependencies Figure 4.7 shows the dependencies between model view types. The upper part of the figure shows these dependencies in the scope of one system element—in this case the system as a whole. Arrows in the diagram indicate a dependency of one view type on another. A depending view is required to be consistent with the view it depends on.

The Interface View specifies the syntactic and semantic interfaces of the system element to its environment. Because of the importance of interfaces in a system architecture, it is the most "public" and outward facing view of a system element. Role View, Data View and Modal View add detail to the specifications of the Interface View defining the high-level internal makeup of the component. The Role View specifies the interacting entities associated with the component and the environment, the Data View defines units of information exchange between these entities, and the Modal View defines the distinguishable operational modes of a component. The Structure View specifies component decomposition and is derived from the Role View and the Interface View. Subcomponents can assume one or more roles. This refinement nature of the structure view establishes roles as abstractions of components. The Service View is a derivative of both the Interface View and the Data View. It partitions the interface according to the services offered. The Interaction View

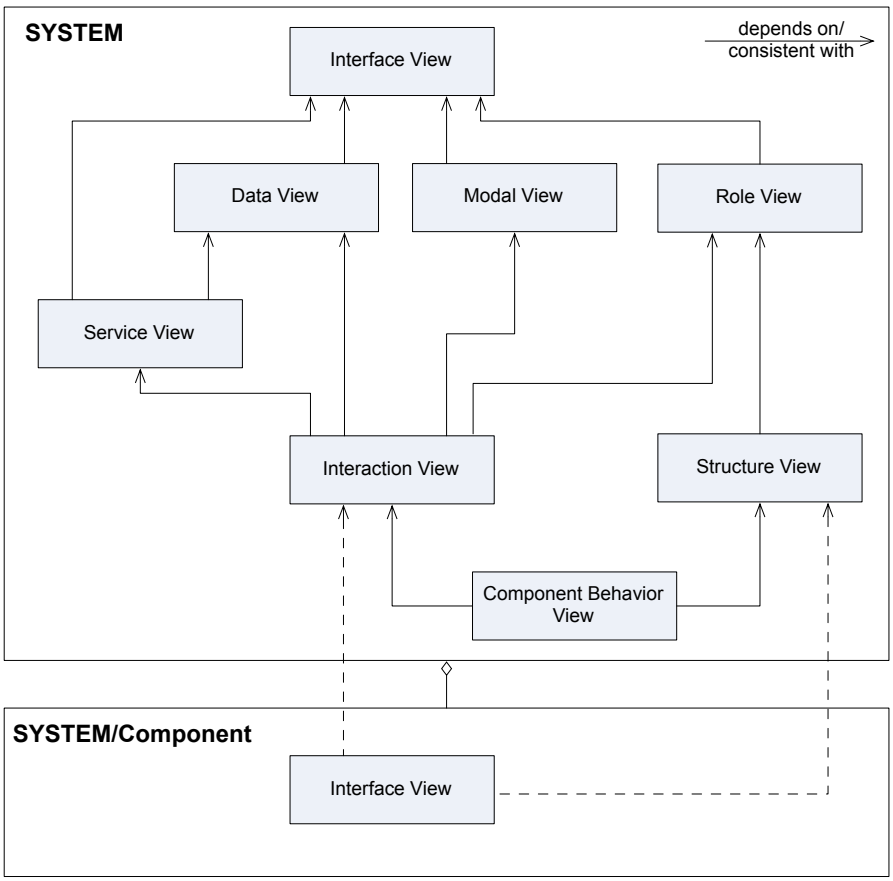

Figure 4.7: Dependencies of logical model views

draws on the Service, Data, Role and Modal Views and specifies interactions between roles in the context of operational modes and exchanged information elements. Each interaction defines one service or part of one. The Component Behavior View represents an aggregate of all the interaction behaviors between subcomponents of a component. It refines the Interaction View and Structure Views. Because of the transitivity of the consistency relations, it is also consistent with the Interface View of a component. This is the most important application of the previously introduced specification–realization pattern. The component's behavior specification is a realization of the component interface, a prerequisite for compositionality.

The views of a component relate to the views of any existing subcomponents. In particular, subcomponent interfaces refine the interaction and Structure Views of a component. A subcomponent's syntactic and behavioral interface must be consistent with the specifications within the views of the supercomponent. This ensures a correct decomposition of the supercomponent and a correct design refinement of the subcomponent respecting all interfaces of its parent.

Logical Model Elements In this section, we provide detailed and precise definitions for the "Logical Model Element". We make use of the notational conventions and formal definitions introduced in Chapter 3 for the description of distributed, reactive systems and their services. Figure 4.8 depicts the elements of the logical system architecture and design model and their relations. As described above, all logical model elements together with the subsequently described technical model elements form the system design model. This model is represented as a collection of model views in applicable notations, realizing development artifacts.

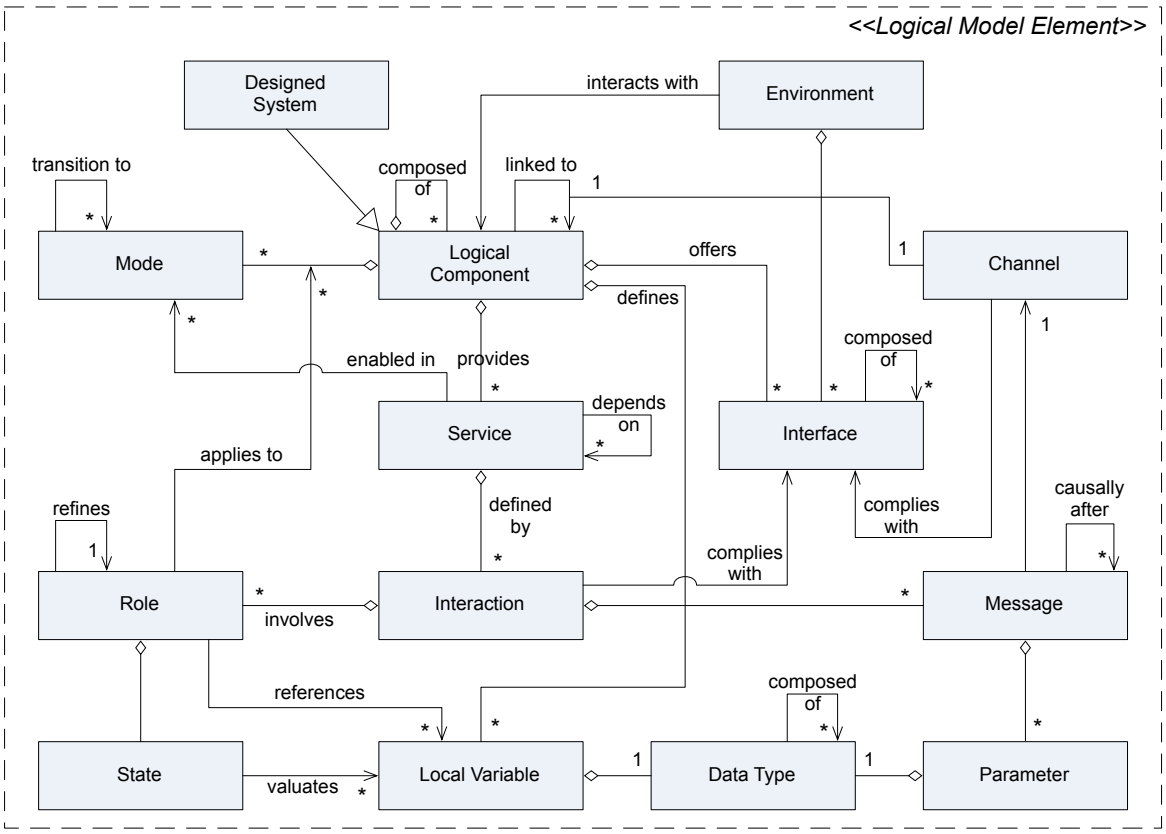

Figure 4.8: Logical model element domain model

The main entity is the *Logical Component*—or just component where the meaning is clear. It represents the fundamental structural building block of a system: a distinct structural unit of the system that fulfills a defined purpose and has clearly defined behavior. Formally, logical components in a system model directly associate with the elements of the set P of components within a system S. Components can be made up of other components. We speak of a decomposition of a component into "subcomponents", with the component itself acting as "supercomponent" or parent component. For a logical component $p \in P$, the set of all subcomponents of p is denoted by P_p. In the same way, all direct subcomponents of the system S are within the set $P_S \subseteq P$.

The *Designed System*—or system in this context—is a logical component that exists once in a system development model; it has no supercomponent ($super.S = \emptyset$) and represents the system and its boundary to the *Environment*, which interacts with the system. The system with its components and subcomponents forms a system decomposition, the main part of the system architecture. Components compose to supercomponents with the top level supercomponent representing the system. We apply the formal definitions of Section 3.3.3 for hierarchical logical system architectures.

From the point of view of the environment, the system acts as one entity. It is expected to perform as specified, while the environment remains outside of direct control. A system design should not make any behavioral assumptions on the environment. It can, however, impose its interfaces as usage assumptions and provide commitments only if the assumptions are fulfilled by the environment. A well behaved component is typically expected to enter a defined error state for all other cases.

A logical component can be *linked to* other components for the same parent component. Each unidirectional link from one component to another component is called a *Channel*, enabling information exchange and communication. Note that we often imprecisely speak of a channel between two components when we actually mean two directed channels in opposite directions. Formally, channels of a system are denoted by set C. The input channels of a component $p \in P$ are denoted by $In(p)$, output channels by $Out(p)$.

A logical component offers a number of *Interfaces* to its environment. This includes a syntactic interface and a behavioral—sometimes called semantic—interface. The syntactic interface precisely defines channels and message types, while the semantic interface specifies the interaction protocol assumed and guaranteed by the component towards the environment. Interfaces can be decomposed into subinterfaces. When a component is refined into subcomponents, its interface remains unchanged. We denote the syntactic interface of $p \in P$ formally by $(I_p \blacktriangleright O_p)$ and the behavioral interface by Com_p.

A logical component provides a number of *Services* to its environment, reflected by the component's interfaces. A service is a modular piece of component functionality with a defined syntactic and semantic access interface. Services can be interpreted as partial functionalities of a logical component that are provided when the access interface is activated. The same component can provide multiple services at the same time. The syntactic service interface is denoted by $(I_F \blacktriangleright O_F)$. The service domain $dom(F)$ specifies the access interface and the service range $ran(F)$ specifies the service behavior in case the access interface was activated, i.e. the input is in the service domain. Services can depend on other services thereby forming a directed, acyclic service dependency graph. In a consistent system model, the access interfaces of all services of a logical component need to be either non-overlapping or the resulting behavior need to be conflict-free. Otherwise, we speak of unwanted feature interaction.

A logical component can define a number of *Modes* of operation. A mode is a high-level "state" of a component that determines distinguishable classes of observable behavior.

Exactly one mode can be active at any given time. Modes can *transition to* other modes according to a defined transition relation for the component. The services of a component can be enabled in all or in certain modes only. This relates to a restriction on the service's access interface and results in different component behavior in that mode. A component's mode applies to all subcomponents.

The behavior of a service is specified by an *Interaction*. In its most abstract case, it is the interaction of the component with the environment, consistent with and refining the syntactic and semantic service interface. An interaction defines the exchange of *Messages* through a partial order over a *causal relation*. Messages are exchanged over channels and have *Parameters* with defined *Data Types*. Concrete message instances provide a valuation of these parameters, thus realizing information exchange. The set if all message types in a system is denoted by M.

For a component with defined subcomponents, the interaction defining a service can be expressed as a collaboration of subcomponents, triggered by messages from the environment, and consistent with the component's interface. The behavior of a service F can be specified, for instance, by a Message Sequence Chart $\alpha \in \langle \text{MSC} \rangle$. This results in a service behavior specification $F = [\![\alpha]\!]$. Here, the MSC resolves to a property indicating valid output behaviors for inputs in the service domain $dom(F)$.

Roles provide abstractions of subcomponents for use in interaction specifications. This enables interaction specification without directly referencing subcomponents as communicating actors. A role represents a communicating entity often intuitively associated with some defined responsibility and behavior. A role may define *Role Guards*, which are *required* conditions that prescribe when a role is applicable. Role guards are expressed in terms of logical components and modes; only a subset of components in a subset of modes can play such a role. Roles can specialize other roles. We will provide a more extensive motivation and definition of roles below. Messages exchanged between two roles constrain messages occurring on channels that link two components playing the respective roles. An interaction specification is a partial ordering of messages exchanged between roles.

Logical components define a number of *Local Variables* with defined data types. Roles reference and modify these local variables. Specific valuations of local variables constitute a role *State*. In contrast to component modes, which relate to high-level control states that remain valid until a defined mode transition occurs, the role state represents a valuation of local variables that may be true for an instant only. The role state is related to the data state of the component.

Roles and thus components can perform local operations in between message exchanges. These local operations can reference and modify local variables. Local operations abstract the actual computations that a component performs.

4.3.3 Technical Architecture and Design Model Artifacts

Figure 4.9 depicts the design model elements dependent on specific technologies and target system environments, targeting system integration and deployment. Together with the logical model elements they form the system design model and similar to these, they can be represented within process artifacts.

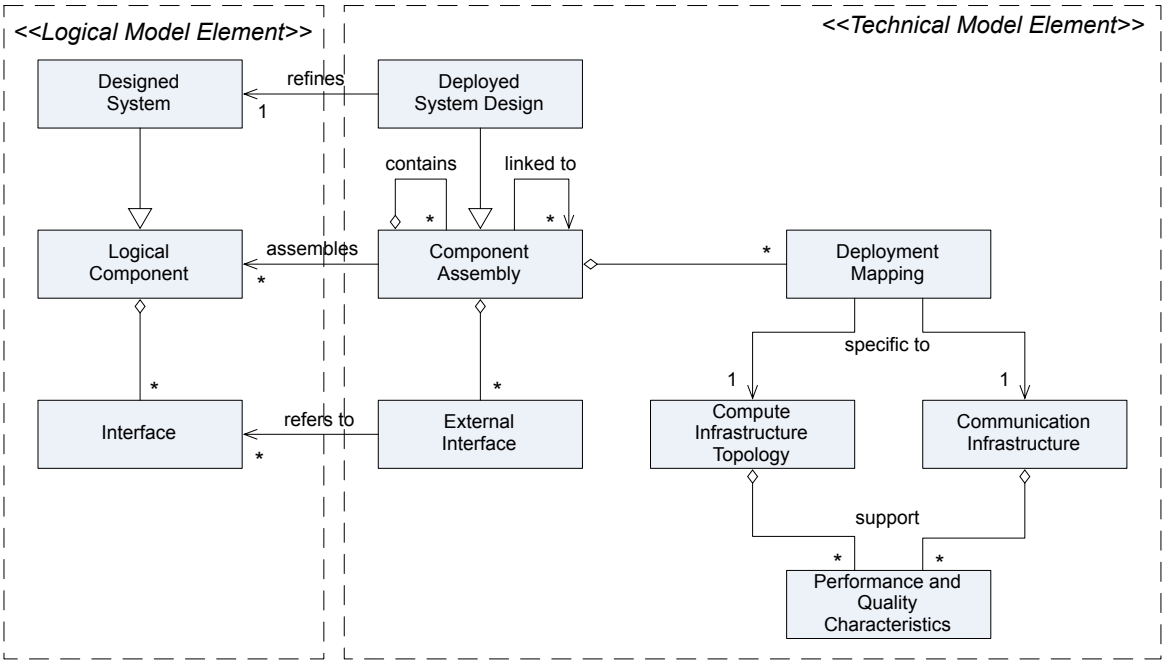

Figure 4.9: Technical model element domain model

We explained the decomposition of logical components into a hierarchical system architecture above. Creating a *Deployed System Design* requires making technology choices based on detailed knowledge of the system's deployment environment with its physical infrastructure made up of computational units—such as servers, embedded control units (ECUs), network processors—and communication infrastructure—such as asynchronous messaging systems, synchronous RPC-style object brokers, and broadcast buses.

A *Component Configuration* specifies the integration of several logical component implementations into a component assembly. It is possible to integrate several identical implemented instances of one logical component specification, for instance for the purpose of high availability and enhanced system performance. All implementations of a component configuration must be specification compatible as defined in Section 3.3.3, as well implementation compatible. The difference between component composition as described above and component integration is the object of the activity. Component composition abstracts a syntactic and semantic interface from two compatible logical components, yielding a specification given two specifications. Component integration defines the outcome

of assembling several implemented components within a target technology or execution environment, forming larger segments of the system. The blueprint for assembly, however, the component configuration, is a design artifact and thus part of the system design model.

Component configurations have an *External Interface* that is the result of integrating the interfaces of all of its components to the environment, as specified. Component configurations can be integrated to larger configurations, just as logical components can be composed to larger components. The deployed system design represents the fully integrated component configuration and specifies the full system integration. It is equivalent in behavior with the specification associated with the designed system but additionally reflects all technology, infrastructure and deployment topology design decisions and targets actual implementation artifacts.

A component configuration provides a *Deployment Mapping* of components to elements of the deployment infrastructure, which consists of the *Compute Infrastructure*—more precisely the topology of how it presents itself as distributed infrastructure to the system—and the *Communication Infrastructure*. Technology choices for compute, storage and communication infrastructure are reflected in this part of the model. The deployment mapping of components and component configurations to elements of this infrastructure must respect these technology choices. All infrastructure choices come with specific *Performance and Quality Characteristics* that further constrain the possible deployments of the designed system. A valid deployed system design provides a complete realization of the designed system within the constraints imposed by technology choices and their performance and quality characteristics, and the deployment topology.

4.3.4 Implementation, Integration and Deployment Artifacts

Figure 4.10 depicts the artifacts created during system implementation and integration. These artifacts represent realizations of the design specifications developed in preceding system development activities, for instance the Component Behavior View of the logical architecture model, with added integration detail in the technical design model. The figure depicts the core artifacts of the implementation, integration and deployment activities. These can be accompanied by numerous additional artifacts specific to the applied technologies and the organizational context.

The *Implemented System* is the fully implemented system targeted at a specific *Physical Infrastructure*. It realizes the deployed system design. Units of deployment are *Deployment Packages*, which consist of or describe elements of the implemented system ready to be deployed on the physical infrastructure. Deployment packages may be composed of other deployment packages, as specified by the integration design.

Deployment packages bundle or reference *Executables*, specified in the deployment mapping. Automated deployment systems may only keep logical representations of the deployment packages, to be assembled during deployment. Current day examples of such systems

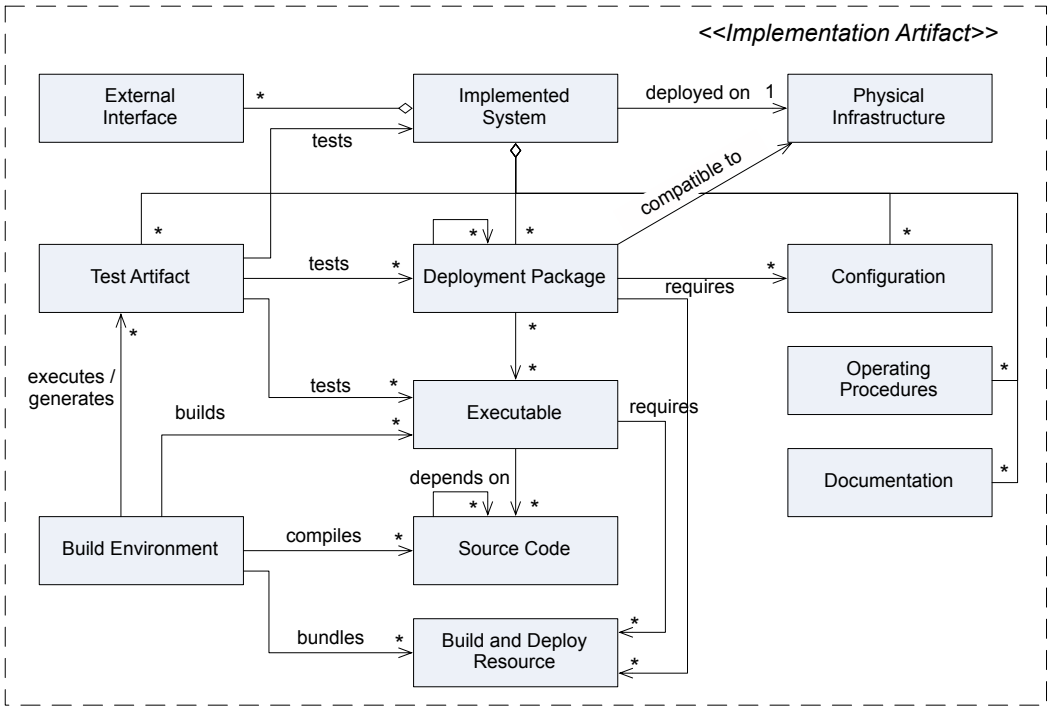

Figure 4.10: Implementation artifact domain model

include Chef[3], Puppet[4] and Apache Maven[5]. Executables can be of any form ready for execution on the target hardware. Executables are created by a *Build Environment* as the result of applying a technology specific build process. Source code and other *Build Resources*, such as software libraries, media files, configuration files etc. are required to perform a successful build.

During deployment and installation of deployable packages, the deployment engineer adds *Configuration* information providing the context for execution in a system environment. Such information can be represented as config files or entries in a config server and may include connection credentials and network addresses to connect to other system elements such as shared databases and name servers. *Operating Procedures* document how operators are expected to install and operation the system. Further *Documentation*, such as user, developer and operator manuals may exist.

Test Artifacts accompany executables and deployment packages in order to ensure and document correctness of build, integration and deployment artifacts. Such test artifacts can be of various kind, such as executable unit tests, integration test drivers with interface stubs substituting for actual hardware, and system test scripts for automated and manual execution. Test artifacts may be executed and generated by the Build Environment.

[3] `http://www.getchef.com/chef/`
[4] `https://puppetlabs.com/`
[5] `http://maven.apache.org/`

Current day examples of build environments include Jenkins[6], Microsoft Team Foundation Server[7] and Buildbot[8], supported by tools such as UNIX/GNU make[9] and Apache Ant[10].

4.3.5 From System Specification to the Implemented System

Figure 4.11 shows dependencies between the main system development artifacts. The implemented system realizes the deployed system design, which in turn is a refinement of the designed system, adding technology specific details and deployment mappings. The designed system realizes the formal system specification, if existing.

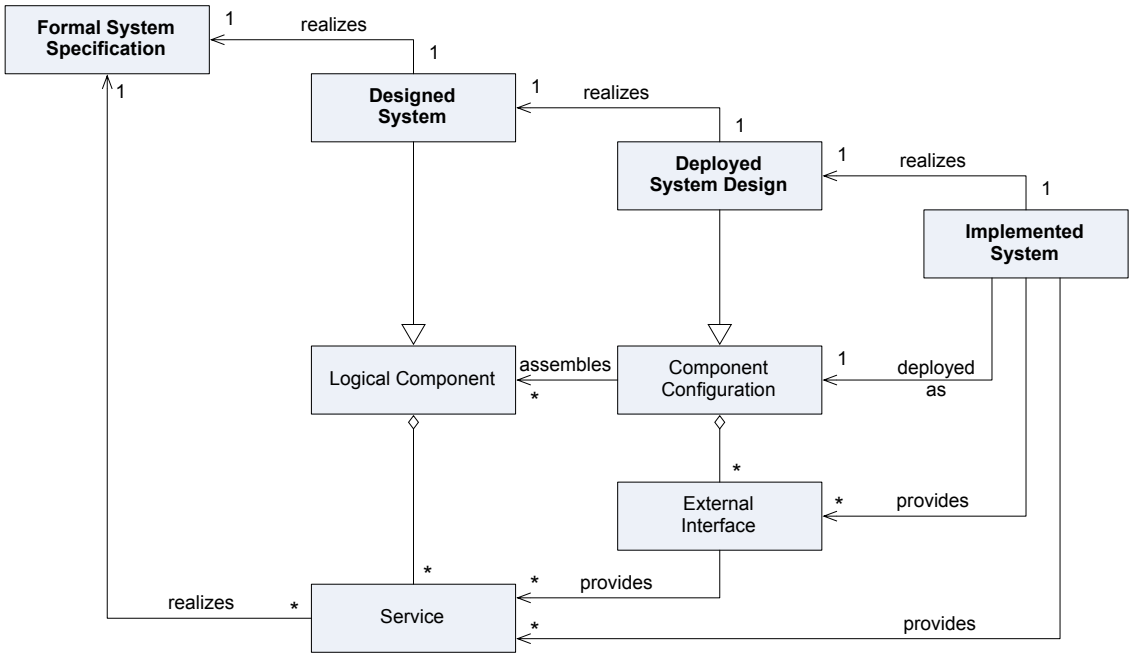

Figure 4.11: System development artifact dependencies domain model

The implemented system realizes the one component configuration associated with the deployed system design. It provides a number of external interfaces to the environment, which consists of users and external systems. External interfaces are a reflection of the services as specified. The way service interfaces are implemented, be it through an interaction of system components or as a monolith, is irrelevant.

[6] http://jenkins-ci.org/
[7] http://msdn.microsoft.com/en-us/vstudio/ff637362.aspx
[8] http://buildbot.net/
[9] https://www.gnu.org/software/make/
[10] http://ant.apache.org/

Properties of Artifacts System development artifacts differ somewhat depending on their abstraction levels and their associated system engineering activities. Figure 4.12 highlights some important dimensions of our artifact model structured by the abstraction levels introduced in Section 4.2.1. We explain these dimensions below.

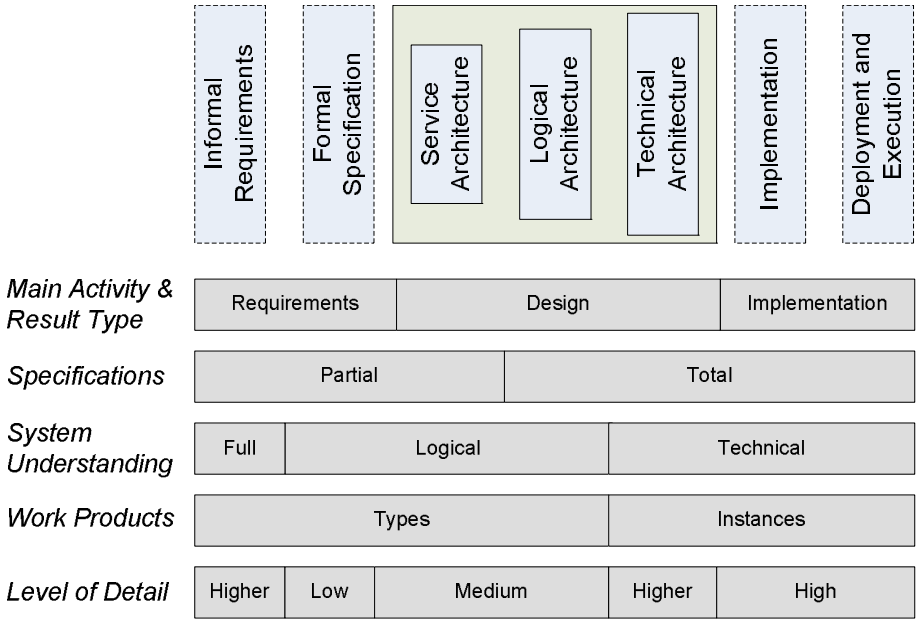

Figure 4.12: Dimensions thoughout development activities

- *Main activity and artifact class:* Requirements class artifacts are produced during the requirements engineering activities and are represented on the informal requirements and formal specification abstraction levels. Design artifacts—in particular the system architecture and design model—are created by design activities and are represented on the three design abstraction levels. Implementation level artifacts are represented on the two lowest abstraction levels.
- *Partial and total specifications:* Requirements to service architecture levels make use of predominantly partial specifications that offer a high degree of specification modularity. Partially specified elements may be easier to manipulate in isolation and easier to combine. Lower abstraction levels use more total specifications. Such specifications are comprehensive with respect to a certain system element and define in detail behavior for all possible inputs and for all structural elements. Component specifications, for instance, are total in that they exclusively and in full represent the behavior of the component.
- *System understanding:* Informal requirements cover all aspects of the system, including functional and technical requirements. The next three lower abstraction levels target the logical characteristics of the system, from its formal properties to

its functions and component decomposition. The lowest layers add technical detail to the artifacts, for instance specifying deployment configurations on specific target environments.

- *Work products:* The higher abstraction levels predominantly define types, while the lower levels represent instantiations of these types. Services and logical components, for instance, are defined as types. Requirements target the different functions of the system independent of their implementation; there are different design model elements for each type of service and component. Their interfaces must be consistent and decomposition follows given rules. On the technical architecture side, an assembly of components can connect multiple instances of the same component type. These assemblies can be distributed and replicated across the network.
- *Level of detail:* The degree of detail information increases with lower abstraction levels. Requirements are an exception: they can cover all aspects of the system and express detailed constraints that are only addressed again at the lowest abstraction level.

4.4 Service-Oriented Architecture and Design Model Views

In this section, we provide a comprehensive walk through the main development process elements related to system design. We explain the views associated with the logical system architecture and design model. We introduce applicable notations and description techniques, provide definitions and examples, and explain the significance of the process elements in a larger development context. We make sure that our model views can represent the concepts of our formal model, enabling an effective representation and manipulation in larger scale real-world projects. We make use of our running example and discuss artifacts and design model elements in the context of the BART system.

4.4.1 ServDL – A Textual Syntax For Service-Oriented Architecture Specifications

We introduce *ServDL* as a textual syntax for the description of service-oriented architectures—the *Serv*ice-oriented architecture *D*escription *L*anguage. It provides precise specifications of design models. A *ServDL* specification can be used instead of a model view's graphical specification; it provides the same definitions. The textual syntax may be easier to process by tools, while graphical notations are typically more intuitive to understand and use by system architects. Also, the enhanced layout possibilities in graphical notations benefit human understanding.

Figure 4.13 shows an example of a *ServDL* specification of a component's interface. Indentation, newlines and bold facing exist to increase readability and to highlight keywords of the domain specific language. They have no meaning to the information represented in the language.

```
Component Interface SYSTEM
  Description
    The AATC system's interface to the environment
  Sub-Interfaces
    Train: <ifspec TrainSpec>,
    Station,
    Communication/Network,
  Channels
    mc_o: Manual Command (Outgoing),
    mc_i: Manual Command (Incoming),
    iu_o: Interlocking Update (Outgoing),
    iu_i: Interlocking Update (Incoming),
    tu_o: Track Update (Outgoing),
    tu_i: Track Update (Incoming),
```

Figure 4.13: BART system interface definition in *ServDL*

As we introduce the different types of system design views in the following sections, we will also provide *ServDL* schemas and examples where useful. *ServDL* schemas are defined in EBNF [Gar03] notation. For instance, the schema for Interface View specifications is defined as:

<ifview> = **"Component Interface"** <identifier> <ifview_details>
<ifview_details> = <description> <ifview_interfaces> <ifview_channels>
<description> = **"Description"** <string>
<ifview_interfaces> = **"Sub-Interfaces"** { <ifview_interface> **","** }
<ifview_interface> = <identifier> [**":"** **"ifspec"** <identifier>]
<ifview_channels> = **"Channels"** { <ifview_channel> **","** }
<ifview_channel> = <identifier> [**":"** <identifier>]

4.4.2 Model View and Component Naming System

Our approach supports a hierarchic decomposition of the system as top level component into subcomponents. We can depict this decomposition as a tree of components with the system component as root. By convention, we name the root component "SYSTEM".

The SYSTEM component decomposes recursively into subcomponents. On each level, the names of the direct subcomponents of a component must be unique. Any component in the system decomposition tree can be uniquely identified by providing its name prefixed by the fully qualified name of the parent component. We call this the hierarchic component identifier. For instance, the subcomponent "Car" of the component "Train" as part of the "SYSTEM" is uniquely identified as *SYSTEM/Train/Car*.

By convention, we denote the environment of the system as ENV. Any substructure that can be identified within the environment such as distinct external systems, subinterfaces, etc. can be arranged in form of a tree similar to the system decomposition into components. The same naming conventions apply.

Elements that are directly associated with a component (as shown in the domain model from Figure 4.8) such as interfaces, services, and channels are uniquely identified for one component. Within the system hierarchy, they are uniquely identified by prefixing the local identifier with the hierarchical component identifier of the containing component.

All information contained in the model is accessible using an OCL [OMG11b] style language, making the model systematically navigable and subject to automatic analysis and processing.

The following list provides examples of model elements and their identifiers:

- SYSTEM : the system, as the unique top level component with fixed name
- ENV : the environment, as a virtual component with fixed name
- SYSTEM.name : name of the system component
- SYSTEM.in : set of channels
- SYSTEM.in.<channel_name> : one specific input channel by name
- SYSTEM/AATC/SSC : a subcomponent of a component of the system

4.4.3 Requirements and Constraints

This section provides more information about the BART system and its parts relevant to this chapter, structured in a way that aligns with the requirements artifacts named in Figure 4.5. The contents of the following paragraphs provide exemplar verbiage for respective sections in the requirements artifacts. In addition to providing more information about BART, this also illustrates the use of these artifacts within the development process.

Scoping The part of the BART case study that is of interest to us is the interplay of the AATC components, between controller, external systems and trains on the track network, as depicted in Figure 1.6. Furthermore, we are interested in describing the behavior of the AATC controller and the controlled trains. We ignore all topics related to routing, communication error recovery and right-of-way signaling.

Constraints The physical infrastructure is prescribed in form of the existing track network, stations, and trains together with the physical communication network, such as switches and gates. Specific constraints are provided. For instance, all tracks are always used in one direction only. The track network has physical properties, geographical dimensions and organizational constraints, such as maximum speeds, and slow-move sections. Some technical systems are existing, performing satisfactorily and require interfacing. Existing systems include the interlocking system for switch and gate control.

Physical and safety constraints limit the system: physical laws applied to the tracks and the trains determine the range of possible operations. After incorporating statistical data and worst case considerations, the operations are limited conservatively to ensure safe operations under all foreseeable circumstances. We use [WKM04] as reference for all physical and organizational constraints that limit the possible range of system design.

Use Cases From the requirements that are present in form of the BART case study document [WKM04], we extracted the following list of use cases:

1. A *train* determines its current status from different sensors in a *consist* (group of cars in a train)
2. A *train* communicates its current status (position, speed, acceleration value) to the responsible *control station*
3. A *control station* receives status messages from all trains in the controlled area in regular time intervals
4. A *control station* receives external input for the controlled area from the *interlocking system* (gate&switch) control and manual speed limit settings
5. A *control station* computes speed and acceleration commands for each *train* in the controlled area
6. A *control station* forwards all commands of an interval cycle to the *VSC* for a reliable safety check
7. The *VSC* relays all safe commands via the comlink to the *trains* in the area
8. A *train* receives a command from its responsible *control station*, and checks the command validity using a time stamp. The train applies the command to all actuators in the consist.

The following sections show design models for the use cases listed above.

4.4.4 Interface View

The *Interface View* provides a high-level view on the system and its components. In particular, it specifies a component's syntactic interface, specifying communication links to and from the environment and exchanged message types.

System Interface to the Environment Analyzing requirements, system constraints, use cases and actors leads to an understanding of the system, its environment and the services provided by the system. Given all existing use cases, it is possible to provide a first rendition of the system and its boundaries with the environment. Use case actors can be categorized into system and environmental actors. Initially, neither system nor environment are structured any further. A viable system substructure is of course required to refine the coarse initial view into a usable design model. This requires steps of system design, which we will show in the following. The environment can be substructured accordingly to the degree additional detail becomes known, for instance the existence of multiple separate external systems. Per definition, the environment is outside the scope of system development; we cannot safely make any assumptions on its behavior. However, we can attempt to describe it as precisely as possible and identify any constraints in system-environment interactions.

Figure 4.14 shows an early version of the Interface View, depicting system and environment together with unidentified communication channels. The type of diagram is a system structure diagram, similar to the ROOM notation [SGW94]. The two main entities represent the system and the environment, respectively. The dashed outline of the environment rectangle indicates that it is an entity outside of the designer's direct control. Lines represent directed communication channels between the system and its environment. Arrows show the direction of the communication but are optional. Little black squares indicate output ports, i.e. sources of communication on a channel, while outlined white squares indicate input ports, i.e. sinks of communication on a channel.

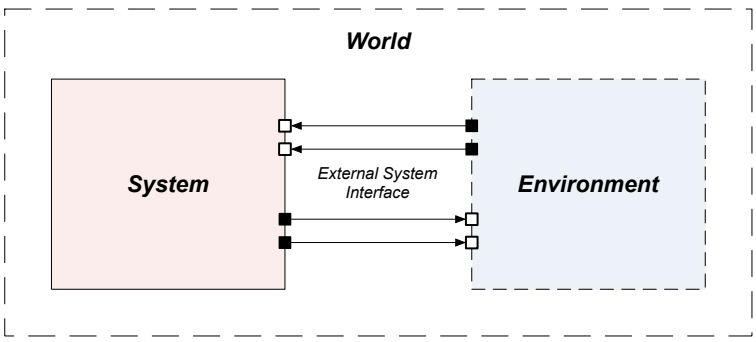

Figure 4.14: System and environment structure diagram

Through high-level system design, by leveraging additional detail from the requirements and use cases, the system designer can subsequently identify the communication channels between the environment and the system. Figure 4.15 shows such a design for the BART AATC controller system. As above, the world is partitioned into the system, under the designer's control, and the environment. The environment stimulates the system through input and receives output from the system. The diagram specifies several named communication channels. It applies a shorthand notation: a labeled communication line delimited on both sides by half filled port symbols indicates two directed communication channels in

opposite directions. The "Manual Command" channels allow human operators interacting with the system to issue directives that override any automatic commands, for in-situ operational decisions. "Interlocking Update" define the channels for the exchange with the interlocking system—primarily signal and switch status provided to the system. The "Track Update" channels allow for operator issued updates of the physical infrastructure, such as tracks and speed limits on track segments.

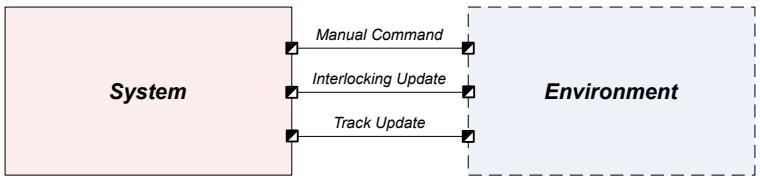

Figure 4.15: BART system and environment

The system's interface can be partitioned into subinterfaces, as indicated by the dashed rectangles in Figure 4.16, a direct refinement of the diagram in Figure 4.15. Subinterfaces indicate distinct identified flows of information between the system and the environment. These subinterfaces may be correlated with separate communication channels, but are not required to. The diagram identifies a "Train", a "Station" and a "Communications/Network" interface on the system side and calls out "Manual Input Interface" and "Interlocking" in the environment.

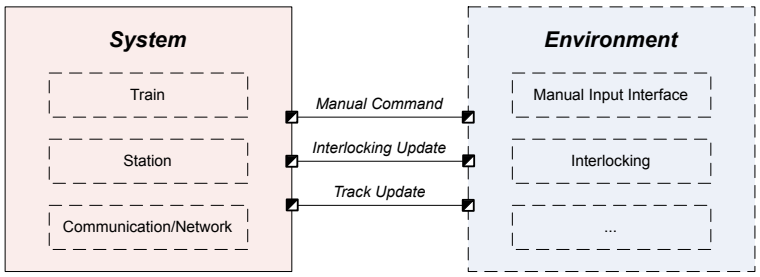

Figure 4.16: BART system and environment with subinterfaces

Component Interfaces The Interface View is constructed similarly for any logical component in the system. It specifies incoming and outgoing communication channels from the perspective of one component and does not show other components. The Interface View always shows the component and its environment—indicated by an entity "Environment" in the structure diagram"—potentially structured into multiple channels and subinterfaces.

Keeping a component's interface generic maximizes the potential reuse of the component design. Specific interactions between entities within a component can be represented in different model views described below, such as the Interaction View.

Detailed Interface Specification Figure 4.17 shows a projection of the logical architecture model element domain model depicted in Figure 4.8, showing only the model entities and associations present in the Interface View, as for instance shown in the BART system structure diagram in Figure 4.16. The logical component and the environment are the two main entities. The designed system is a special case of a logical component. Component and environment both reference interfaces. Interfaces are associated with channels. An Interface View can represent multiple interfaces between system and environment. We speak of the system interface partitioned into subinterfaces.

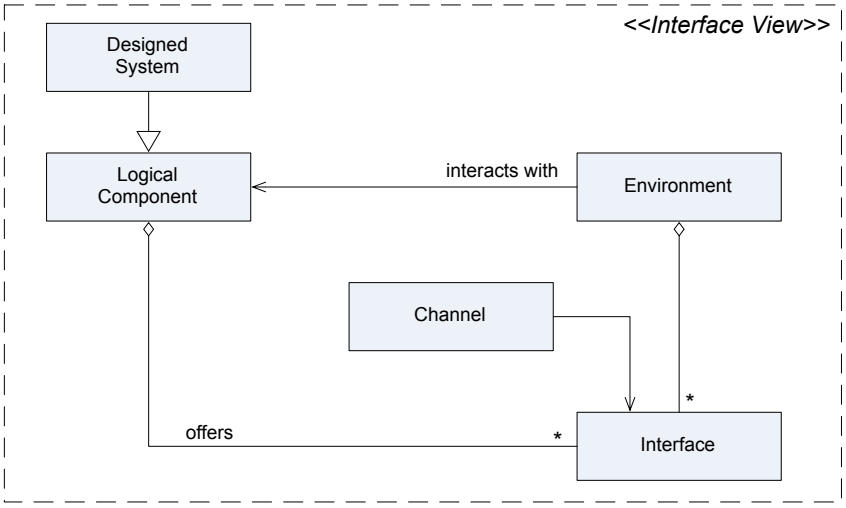

Figure 4.17: Interface View domain model

Besides their graphical representation in form of a system structure diagram, the interfaces and communication channels can also be precisely defined in a textual syntax and by formal specification. Figure 4.13 above shows the definition of the interface of the SYSTEM in *ServDL*. In this specification, all channels are individually named and associated with an additional short alias.

Additional details can and need to be added to the Interface View in further refinements of this model. This includes precise definitions of the syntactic and behavioral component interfaces. Our formal system model based on FOCUS supports such formal specifications, but any other sufficiently precise language will do as long as an interpretation in terms of our formal semantics exists.

Figure 4.18 shows an exemplar interface specification in FOCUS for the BART "Network" component, with bidirectional channel pairs providing reliable, ordered information transport with unspecified time delay. The time abstraction operator was defined in Section 3.1.2.

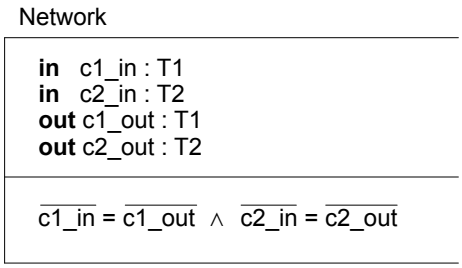

Figure 4.18: Network component interface specification in FOCUS

4.4.5 Service View

The *Service View* specifies the behavior of a component as a set of separate services. It represents a refinement of the Interface View in that it decomposes the component's syntactic and behavioral interfaces into services with a service access protocol for each service. The service access protocols get specified as interactions of the component with its environment. We use the MSC notation—defined in Section 3.3.2—to specify such interaction patterns.

Figure 4.19 shows an exemplar service specification using the MSC notation. It depicts the "Interlocking Data Exchange" service provided by the system to the environment. The environment initiates the service by sending an "update" message with "interlocking data" content. The system responds by sending an "update" message with "train data" content. While being an extremely simplified specification of a service, it shows the interacting entities, the sequence of communication messages exchanged and the types of messages exchanged.

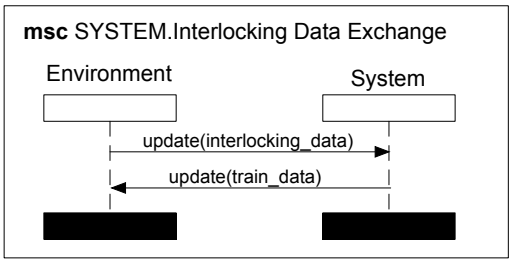

Figure 4.19: BART system service "Interlocking Data Exchange"

The Service View may also identify services internal to the component, for instance representing continuous or recurring behavior with no external trigger and no external output. Figure 4.20 shows the "Provide Train Commands" service as an example of an internal system service. It is triggered by a timer state condition and performs a sequence of internal operations related to computing, validating and communicating train commands. No communication with the environment occurs. These and all other services will of course

be refined in subsequent views and may be realized as complicated interactions between subcomponents. The responsibility of the Service View is to identify all services by name and to specify the service access protocols for services provided to the environment.

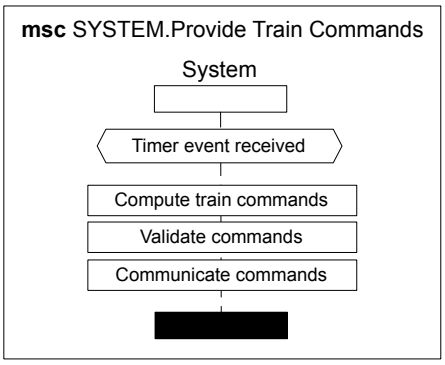

Figure 4.20: BART internal system service "Provide Train Commands"

Figure 4.21 shows a projection of the logical architecture model element domain model depicted in Figure 4.8, showing only the entities and associations that are present in the Service View. Logical components provide services defined by interactions, consisting of causally related messages on communication channels. The service interactions must comply with the interface specifications of the Interface View.

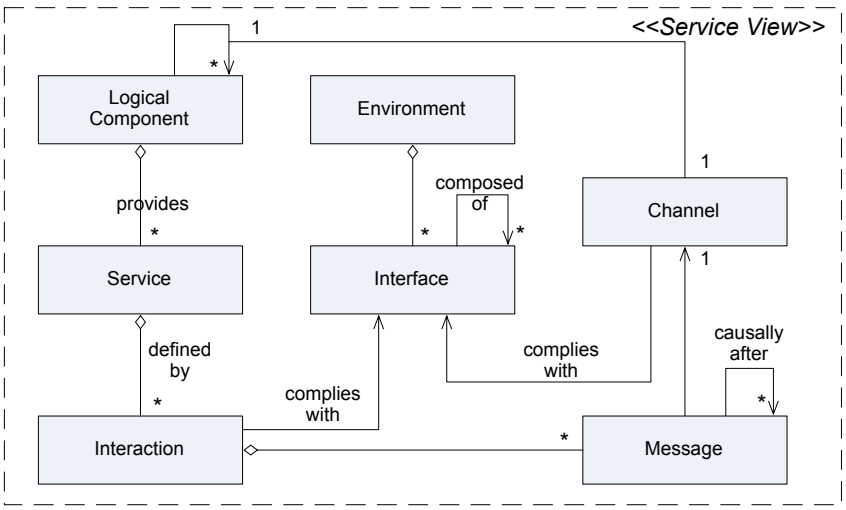

Figure 4.21: Service View domain model

Figure 4.22 shows an abbreviated *ServDL* specification of the BART system services. It refers to the specification of MSC interaction patterns for each service. We will show further MSC specifications when we introduce the Interaction View, below.

```
Component Services SYSTEM
  Description
    The AATC system's services, as refinement of its
    interface to the environment. Note that not all
    services are enabled at all times.
Services
    System Startup,
    Interlocking Data Exchange,
    Manual Track Update,
    Station-Train Communication,
    Train-Station Communication,
    ProvideTrainCommands,
    ReceiveTrainStatus,
    Emergency Detection,
    System Shutdown
Service Specifications
    System Startup: <msc System Startup>,
    …
```

Figure 4.22: BART system services *ServDL* definition

4.4.6 Modal View

The hierarchical decomposition of a system into components and subcomponents as described above targets the design of system structure. Another dimension of system design is refining the behavior of the system or a component. A seamless development approach will support the designer in constraining the behavior on a high level first, before refining it to a detailed behavior specification. This keeps designs open and flexible, while allowing subsequent seamless refinement.

Our approach offers the system designer the ability to constrain system and component behavior using operational modes, specified in the *Modal View*. The concept of operational modes—also known as high-level system states—has been applied successfully in the design of distributed reactive systems, e.g. for automotive control units [HKB09, BBR+07, Sch06b, MR03]. Components assume exactly one operational mode at one time. The current operational mode influences the observable behavior of the component. The operational mode can change in reaction to received messages or internal conditions. Not all components exhibit observable mode changes; in this case, we assume that they define one operational mode that never changes.

Figure 4.23 shows a projection of the logical architecture model element domain model depicted in Figure 4.8, showing only the entities and associations that are present in the Modal View of our system design model. Logical components define a number of modes. Transitions exist between the modes of a component.

The system designer has the possibility of identifying a component's modes and their transition relations, thereby realizing high-level behavioral design decisions. The succession of a component's modes can be seen as a behavioral wireframe. Subsequently, the designer can associate specific behavior with each mode of a component. We will describe this below when discussing the Interaction View. A benefit of the Modal View is that new operational

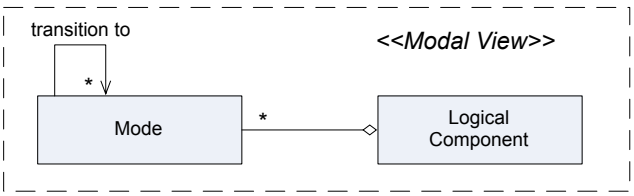

Figure 4.23: Modal View domain model

modes can be added to a design model without changing existing specifications. Existing detailed designs can be associated with the new operational mode, increasing the potential for design reuse.

A component begins its operation in its initial mode and can switch modes over time as permitted by the available mode transitions. The exact condition or event leading to a mode change is not specified in the Modal View. If a component defines operational modes and decomposes into subcomponents, then the operational modes also apply to each subcomponent. Subcomponents may define their own operational modes, as necessary. It is up to the system designer to constrain the possible combinations in order to maintain a manageable system design. The behavioral interface of a component does not change when refined into subcomponents.

Component Mode Diagram We make use of a state machine based notation to capture a component's modes and their transitions. A component mode diagram (CMD) contains a label **cmd** indicating the type of diagram, a unique name identifying the diagram and the component of the system it is associated with, and the depiction of a mode automaton. By convention, the full name of the diagram is constructed as `Component-Identifier:Modes`, for instance `SYSTEM/Train:Modes`.

We define a *mode automaton* A as a 3-tuple $A = (M, \delta, m_0)$, where the elements of the tuple denote

$$\begin{array}{lcl} M & : & \text{a finite set of modes,} \\ \delta \subseteq M \times M & : & \text{a (nondeterministic) transition relation,} \\ m_0 \in M & : & \text{the initial mode.} \end{array}$$

A mode automaton has a very simplistic representation. It provides a set of modes M and transitions between modes, but does not label the transitions and modes with inputs and outputs. This reflects that it does not provide any specific definition of conditions that define modes or trigger mode transitions. This information will be provided within other model views.

The following consistency rules apply to all CMDs:

- All mode names must be unique within the automaton

- All mode names must be unique within the component and all subcomponents
- All modes must be reachable from the initial mode

Figure 4.24 shows the component mode diagram for the top level SYSTEM component of our running example. It defines the four modes “Maintenance”, “Manual”, “Automatic” and “Emergency” and their possible transitions. “Maintenance” is the initial mode.

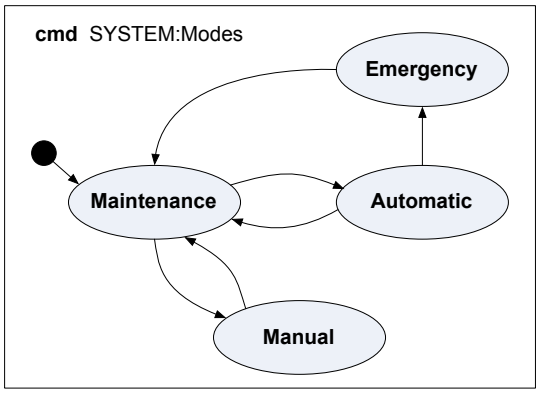

Figure 4.24: BART SYSTEM modes

A component mode diagram has a *ServDL* representation. Figure 4.25 shows the *ServDL* mode definitions for the diagram in Figure 4.24.

```
Component Modes SYSTEM
  Initial Mode Maintenance
    Description System is operational, providing maintenance access and reports.
      All trains are stopped.
    Transitions To Manual, Automatic
  Mode Automatic
    Description System is operational, with fully automatic control of train operation.
    Transitions To Maintenance, Emergency
  Mode Manual
    Description System is operational. Trains operate only by manual control on low speeds.
      Emergency detection is activated.
    Transitions To Maintenance
  Mode Emergency
    Description Emergency situation detected, needs resolution.
      All trains brought to emergency halt.
    Transitions To Maintenance
```

Figure 4.25: BART SYSTEM mode specification in *ServDL*

The following schema applies to mode specifications in *ServDL*:

<comp_modes> = **”Component Modes”** <identifier> { <mode> }
<mode> = [**”Initial”**] **”Mode”** <identifier> <mode_details>
<mode_details> = **”Description”** <string> [<mode_transitions>]
<mode_transitions> = **”Transitions To”** <identifier> { ”,” <identifier> }

Difference Between Modes and States We distinguish between operational modes and component states as defined in Section 3.1.3. We will discuss component states in more detail below in the context of the Interaction View. Our interpretation follows [MR03]: we interpret states as a concept of executing a program or state-based behavior specification; states express explicit valuations of local variables of a component. A program assumes a certain state at a certain instant in time and continues with the computation which yields a sequence of successor states. Depending on the granularity of the state definition (control state vs. data state), the actual component states change more or less frequently.

Operational modes on the other hand realize a partitioning of a component's behavior according to certain high-level conditions. Modes persist over a certain period of time and constrain component behavior on a high level of abstraction. Modes can be seen as an abstraction of execution states, by defining a partition of the state space into sets of states with similar observable behavior.

In our approach, we consider the existence of a fully specified deterministic state machine, defining a component's total behavior as an implementation artifact that is the result of the application of a mostly automatic behavior synthesis process. A service-oriented system specification, consisting of a component's partial pieces of behavior is translated into a total state machine after checking consistency, completeness, implementability and correctness, by applying a closed-world assumption to the previous open-world design model. These steps carry the design model to an implementation, reducing the level of abstraction. Such an implementation can now be easily translated into executable code, using code generation techniques and tools.

In addition to modes, we also provide the notion of state on the modeling level through state labels for roles and components in service specifications. This enables adding detail regarding the flow of execution to service specifications, where necessary. Because of the partiality of service specifications, such state markers are always instantaneous assumptions of certain (partial) execution states.

The modes and state markers of the design model will result in a set of states and transitions forming a state machine implementation. The structure of this state machine will commonly not show a high similarity with the flow and names of states of the design model. Named and labeled states and modes of the design model will not appear equally in the implementation due to the synthesis algorithm with implicit product-automaton constructions, optimization and minimization steps. In practice, slight modifications to the service specifications will often result in quite significant changes of the resulting state machine.

This interpretation of modes as abstract states over time also follows the reasoning of Gurevich. When defining the abstract state machine formalism [Gur00], he sees states as full instantaneous descriptions of the algorithm. When using states in a more restricted sense, interpreting them as expressing control over time, he suggests to use the term mode instead. While Turing machines may have an infinite state space, their control and the number of configurations is finite, cf. [Gur00].

Related Work Maraninchi et al. [MR03] have observed that the notion of *running modes* appears frequently in real-time embedded programs, for instance in the avionics application domain. Formal synchronous domain-specific system design languages, such as Lustre [CPHP87] are particularly well suited for designing such systems. Lustre has a dataflow oriented declarative style of specification. Maraninchi et al. propose an extension of Lustre with mode automatons.

Without explicit constructs for the design of systems with modes, such running modes would often be specified using conditional (if-then-else) control structures with a predicate identifying the mode in the conditional expression. This leaves the modes implicit and leads to redundancy within specifications. *Mode automata* promise to improve on readability and maintainability by introducing an explicit notation based on a formal semantics for modes that can be associated with synchronous computations. The authors define a trace semantics for mode automatons and define parallel and hierarchical composition operators.

Maraninchi et al. propose a criterion for a construct that supports system design using modes: "it should be possible to project a program onto a given mode, and obtain the behavior restricted to this mode" [MR03]. Their mode automata fulfill this criterion. This is similar to the possibility of projecting concurrent behavior onto one component, a more commonly supported concept. The characteristic with mode automata is that only the transition function changes when a mode changes. The local data and memory of the component remains unchanged; it is global to all states.

Labbani et al. [LDR06] provide an application of these mode automata embedded into the Scade (Safety Critical Application Development Environment) [Est04] engineering approach. Scade is based on the Lustre language, but also contains state-based modeling techniques known from Esterel [BG92]. It provides modeling facilities with block diagrams for continuous control and state machines (called Safe State Machines, SSMs) for discrete control. Both techniques allow for hierarchical refinement. Scade, however, lacks a clear methodology for the separation of data-flow and control-flow modeling parts.

The authors show the application of their approach using a case study from the automotive domain [LDR06]. They show different outcomes for designing a system with a clear separation of control and data-flow, and without this separation. Having a methodology for separating control and data-flow and supporting constructs, such as mode automata increases readability of the models as well as maintainability and reuse of functional blocks. Minor modifications to the behavior of certain modes are contained in a fraction of the model and do not spread across the entire behavioral model. Ambiguity is reduced.

A control and data-flow separation approach with modes is well suited for real-time embedded systems that are mostly regular but can switch behavior depending on observable conditions. Most bigger embedded systems show both continuous control and reactive behavior parts. An approach that integrates both parts seamlessly based on a common formal foundation helps system development. Alternatives, such as multiparadigm approaches, where modeling elements of multiple approaches are combined, or translation

approaches, where one type model is transformed into the other type and then refined are either too complex to handle or require too much knowledge and modeling discipline from the system designers.

Bauer et al. [BBR+07] use modes in automotive software engineering. Their approach provides a domain-specific integrated development approach that covers both control and reactive systems in the automotive domain, by extending existing approaches and tools. They provide modeling facilities for system structure, data-flows, mode transitions, state transitions and architecture modeling. All models are based on the semantic model of FOCUS/AutoFocus, are integrated and provide methodological support across multiple abstraction levels. Mode transition diagrams (MTD) are automata where transitions are labeled with conditions. Associated with different modes can be data-flow diagrams (DFD), system structure diagrams (SSD) and state transition diagrams (STD). STDs are automata that specify the behavior of an atomic component (not decomposed further). Transitions are labeled with communication events and component activities. In general, MTDs are used for coarse grained system design, while STDs provide fine-grained details for component behavior.

Schätz [Sch06b] provides an approach for model-based engineering of embedded control software that includes modes for control parts and reactive parts of systems.

4.4.7 Role View

In the Role View, roles get defined as first class entities representing actors in a component's interactions. Figure 4.26 shows a projection of the logical architecture model element domain model depicted in Figure 4.8, showing only the entities and associations that are present in the Role View. A role can be associated with a logical component in specific operational modes. The logical component can either be the component itself or any of its subcomponents.

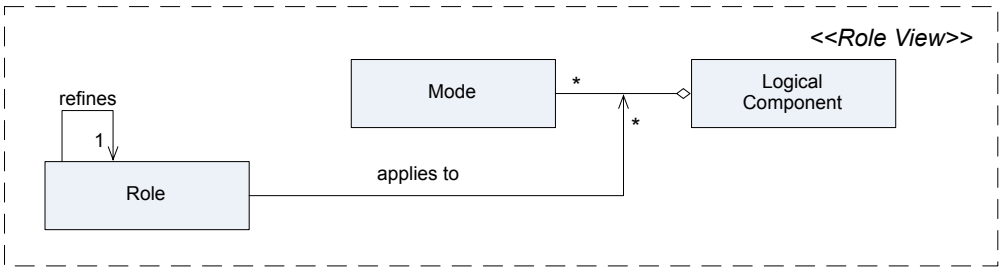

Figure 4.26: Role View domain model

During structural and behavioral refinement of a component, modes and roles should be designed first, before creating specific interactions and subcomponents. Roles can be interpreted as abstractions of subcomponents with clearly defined functional responsibilities.

They exist before the component substructure gets defined. Component behavior can be defined in the Interaction View based on roles and operational modes. One of the primary benefits of roles is to avoid direct reference of logical subcomponents in the specification of interaction patterns defining component behavior. Doing so would prevent a reuse of these interaction patterns across multiple similarly structured components and would require an early mapping of responsibility in interaction patterns to components. Brittle design models would result that are not robust to changes in a component's substructure. Such changes occur frequently while refining the design of larger components.

If the component defines operational modes, then roles can apply to a subset of these modes to provide additional modularity. This reflects the idea that components can change their observable behavior as operational modes change. Certain roles and their associated behavior only exist in certain modes.

Once a component's substructure gets laid out, roles can be mapped to subcomponents. The Role View specifies this mapping. One role can be mapped to one or multiple subcomponents and multiple roles can be mapped to the same subcomponent. If roles are defined for certain operational modes only, then the mapping is only effective for these modes. This flexibility in mapping makes roles very versatile elements of system design, suitable for many types of systems and application domains. The system designer has substantial freedom in defining roles and their interaction behavior, without constraining subsequent component decomposition too much. The design model, however, can get very complex when many roles get combined with many operational modes. It is up to the system designer to find simplifying design patterns and architectural solutions in order to keep the system design and the component complexity manageable. If the number of operational modes is high, then the number of roles and subcomponents should be kept small; alternatively if there are many roles then the number of operational modes should be limited.

Methodological Use of Roles Roles represent abstract entities in component behavior specifications with clear functional characteristics. The subcomponents of the component implement (play) these roles—but not necessarily at all times. In general, a role applies to a number of logical components in a number of modes; more precisely to a set of combinations of component and mode. In the following, for reasons of notational simplicity, we interpret the current component as system and any subcomponents as the components of the system. Set identifiers can be constructed to match the hierarchical decomposition of the system.

Formally, if C is the set of logical components in a system S and M is the set of modes that this system can assume, a role $r \in R$ within the set of roles R is defined as

$$R \to \mathcal{P}(C \times M)$$
$$r \subseteq C \times M$$

An example can illustrate this concept. We know that there are clients and servers in our system and we can precisely define the interaction pattern that occurs between two such

entities. Naturally, the responsibilities of client versus server are clearly enough delineated to lend themselves as candidates for roles. The interaction between both can be represented an exchange of messages between the communicating entities.

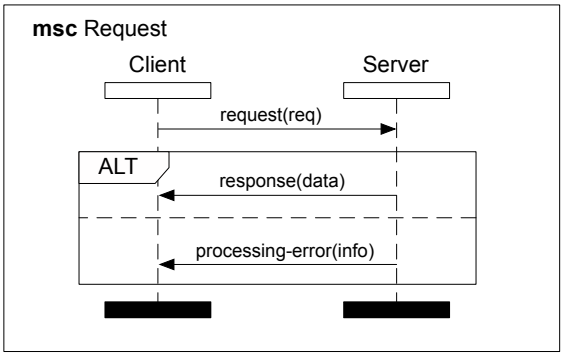

Figure 4.27: Interaction pattern for a Client-Server interaction

Figure 4.27 shows such an example interaction using a sequence diagram. Without explaining the specific semantics at this point, we get an understanding of the system dynamics. The *Client* role sends a message to the *Server* role containing a *request*. The server performs some internal computation. The expected result is a *response* message back to the client. In case the server computation fails, a *processing-error* message is relayed back to the client.

We can apply the roles of client and server as well as the associated interaction pattern to a concrete system design situation, such as our BART example. At an early time in the design, we do not exactly know the components in our system that will be clients and servers. Furthermore, some clients might temporarily act as servers and vice versa. After the system designer decomposed the system into subcomponents—specified within a component decomposition diagram—we can now identify which components should play which role.

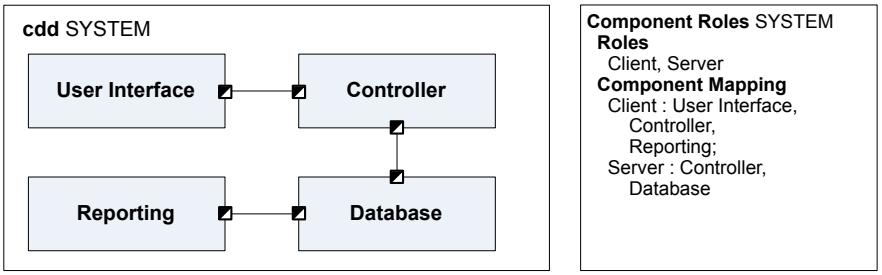

Figure 4.28: System decomposition CDD and *ServDL* mapping of roles to components

Figure 4.28 on the left shows an exemplar decomposition of the system into components, connected by channels. It also shows a mapping of roles to components, expressed in

ServDL, on the right. This role mapping constrains how these four components can behave in case of request and subsequent response or error messages. This is of course not a comprehensive specification of the shown components —each of the four components still needs further behavior specifications and additional roles mapped to expose their intended behaviors. We have also not yet defined how the Controller component can play both roles of a client and of a server simultaneously.

Figure 4.29 introduces a further refinement. For our simplified purpose, imagine that the Controller component has two operational modes: in the "Accepting" mode, it receives requests and acknowledges them with a response; in the "Processing" mode, it stores the received information in the Database. The mode diagram (CMD) shows how the two modes of the Controller are related and can change. On the right side of the figure is a *ServDL* role definition with a mapping to components. Note that now the Controller only performs the server role in its Accepting mode and only acts as a client in its Processing mode.

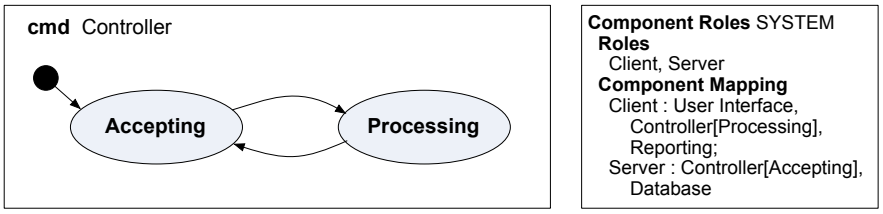

Figure 4.29: Operational modes CMD for Controller and refined *ServDL* mapping of roles to components in modes

This highly simplified example shows the potential of the role concept in its combination with component mapping subject to operational modes. In the following, we will explain further methodology related to our approach, as well as notations and formal semantics associated with all model elements.

Role Definition and Component Mapping The roles are defined and mapped to components in *ServDL* following the schema:

<comp_roles_def> = **"Component Roles"** <identifier> <roles> <mappings>
<roles> = **Roles** <role> { **","** <identifier> }
<mappings> = **"Component Mapping"** { <mapping> }
<mapping> = <identifier> **":"** <identifier> { **","** <identifier> }

There exists no graphical syntax in our approach to define roles and map them to components. If only graphical specifications are used, then roles can be treated as defined if they exist within an Interaction View for a component. The role mapping to subcomponents need to exist explicitly, preferably in *ServDL*.

Related Work Caetano et al. [CZST05] apply roles to business process modeling. They provide an approach relying on two main design models: The business object model describing the components of a business process and the role model describing the interactions between components and any constraints. Roles can be bound to business objects to provide context to their behavior. This approach is similar to our approach in the goal of separating roles from components and providing a subsequent mapping. Our approach provides further model views for various behavioral dimensions of a component or the system; in addition, our approach is independent of any application domain. We explicitly concur with the authors in that the "value of using role modeling increases with the need of making explicit the patterns of interaction between business objects" [CZST05].

Roles are applied in the Reference Model for Open Distributed Processing (RM-ODP) [AG07]. Here, roles are related to object interfaces and to relationships between entities. In particular, one definition defines role as "Identifier for a behaviour, which may appear as a parameter in a template for a composite object, and which is associated with one of the component objects of the composite object" [AG07]. This is closely related to our definition of role and their methodological use when refining the structure of a component design, relating them in interactions to other roles and subsequently mapping them to child components of the composite component or system.

Roles have also been applied in the modeling of multiagent systems. For instance, [FG98] defines a role as "an abstract representation of an agent function, service or identification within a group. Each agent can handle several roles, and each role handled by an agent is local to a group". This relates to our approach in that agents represent active, interacting child components of a parent component called a group. The actual system entities are agents and roles are an abstraction to be mapped to agents.

4.4.8 Interaction View

The *Interaction View* enables the system designer to precisely specify interaction behavior without explicitly requiring an existing decomposition. As explained above, the Interaction View leverages defined roles and operational modes. Figure 4.30 shows a projection of the logical architecture model element domain model depicted in Figure 4.8, showing only the entities and associations that are present in the Interaction View.

The services of a component are defined by interactions. An interaction is defined as a number of roles exchanging messages arranged in causal sequences. Messages can be parameterized and roles can assume execution states. Roles apply to combinations of subcomponents and operational modes. The messages defined within the interaction need to be consistent with the channels defined for the component and its subcomponents.

Interaction pattern specifications We make use of our extended MSC notation as specified in Section 3.3.2 to represent interactions in the Interaction View. This notation

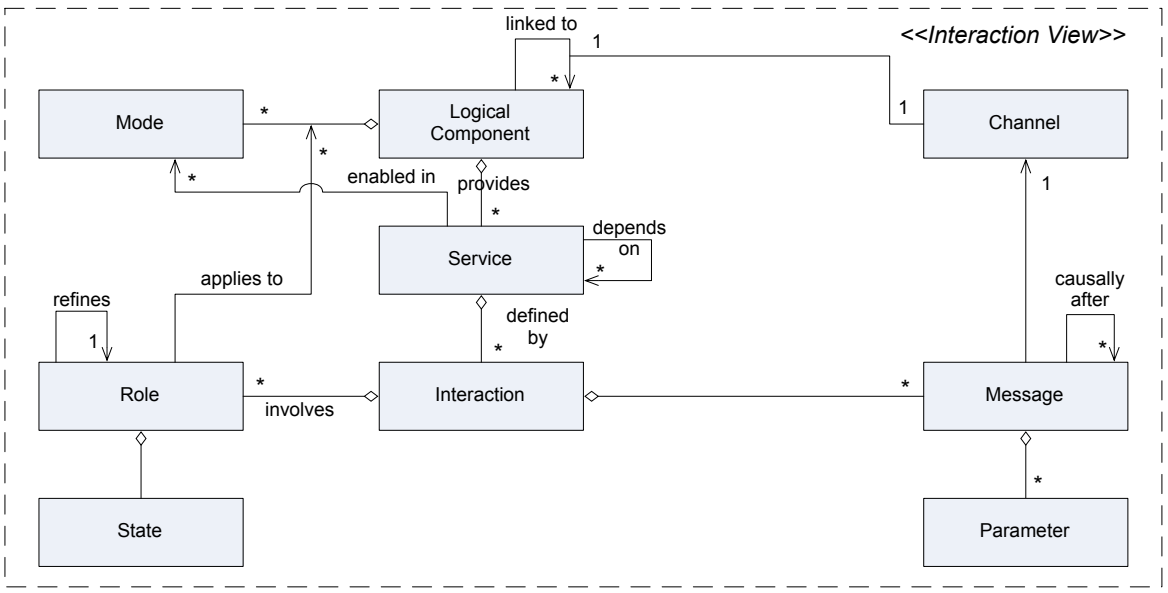

Figure 4.30: Interaction View domain model

provides Basic MSCs for the specification of interaction patterns between distributed system entities. We make use of such MSCs to define interaction patterns between roles that have been defined in the Role View.

High-Level MSCs (HMSCs) provide a means to structure MSC-based specifications using a flow-like syntax. Figure 4.31 shows exemplar interaction specifications for the BART system in form of HMSCs. The HMSC on the left specifies a sequence of three interactions for the system. The fact that the HMSC is labeled as SYSTEM identifies it as the master behavior specification for the system level. Other behaviors are referenced from within. HMSCs or Basic MSCs will be required for each of the three referenced interactions. This sequencing of interactions means that the system goes through three subsequent distinct phases and behaviors: “Startup”, “Operation” and “Shutdown”.

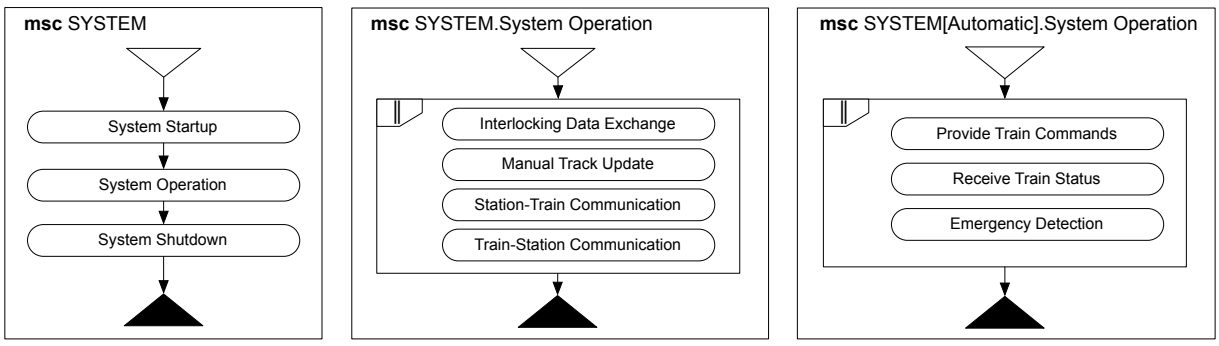

Figure 4.31: BART system interactions, defined as HMSCs

The HMSC in the middle defines one of the referenced interactions on the system level. It also shows the parallel composition operator. All of the four referenced interactions execute concurrently. This means that the system exhibits all four behaviors. The four referenced behaviors can be specified independently in separate interactions. The HMSC on the right defines further behaviors as part of the "System Operation" interaction. In this case, the referenced three parallel composed behaviors are only enabled if the system is in mode "Automatic", as indicated in the title label of the diagram.

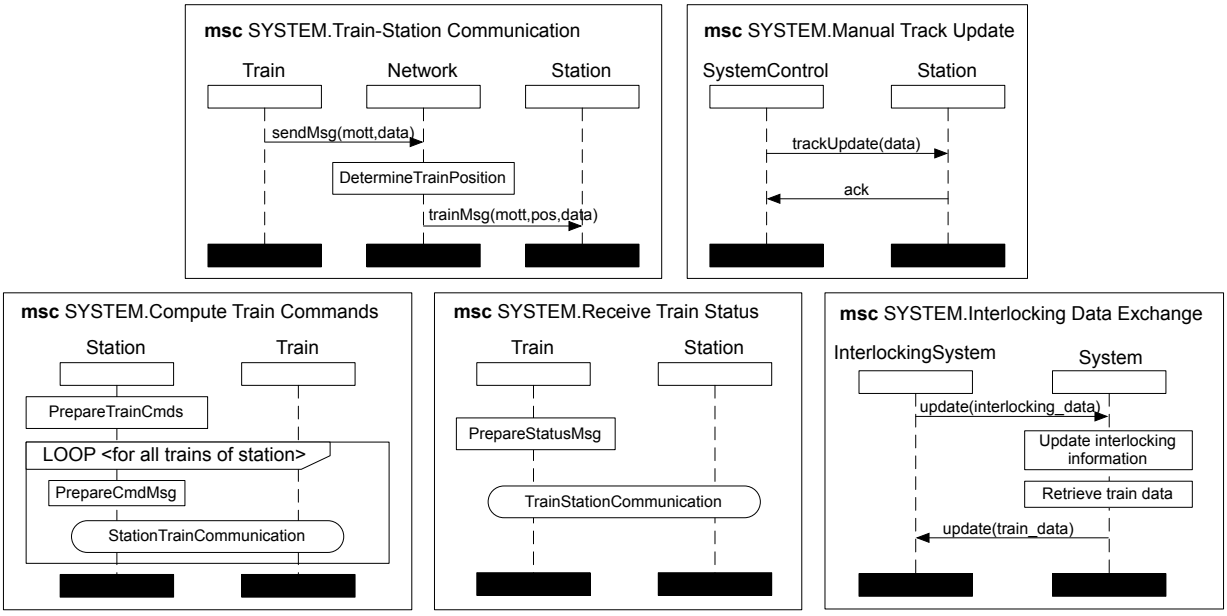

Figure 4.32: BART system interactions, defined as Basic MSCs

Figure 4.32 provides specifications of further system interactions as Basic MSCs. The interactions realize the individual services of the system, as defined in Figure 4.22. The first service specifies a communication from the Train to the Station via the Network. The Network performs the local action to determine the train position before forwarding the message. The second and third services show interactions between two system components. The fourth and fifth service show control structures, such as the conditional LOOP operator and references to further interaction specifications.

Interactions in Basic MSCs specify role changes [KM04b] and operational mode changes. A role change is indicated by a subsequent role actor definition on a role actor axis, as part of the interaction sequence. From this point on, the described interaction behavior applies to a different role or role in an operational mode. It is best to use this notation sparingly to limit specification complexity and confusion with the reader. A mode change is indicated by a state label across all role axes indicating the new mode. This transitions the component and all subcomponents to the new operational mode. The mode transition must be permitted by the mode transition automaton specified in the Modal View. For a discussion of the differences between modes and states, see Section 4.4.6.

Methodological Use of Interactions Interactions provide precise, detailed behavior specifications. The MSC notation we apply provides formal semantics and a clear interpretation defined in FOCUS and the formal model introduced in Section 3.3.2. In our Interaction View, we use roles instead of components, for the axes of the interaction MSCs. This is possible because we subsequently and unambiguously map roles to components.

A textual syntax exists for MSC specifications as referenced in Section 3.3.2. We do not provide a *ServDL* syntax for the Interaction View because the MSC based interaction specifications are comprehensive mapping interactions to components.

The messages in the Interaction View are identified by name within the context of a service interaction specification. The detailed structure of the messages can be left open when the interactions are initially specified. Subsequently, argument lists can be added and detailed message type specifications can be provided in the Data View, described below. Similarly, interactions can reference role state, e.g. as the consequence of an interaction or as a trigger for an interaction. The data state vector for each role or the component as a whole can similarly be specified in the Data View.

Related Work Alternative notations for the specification of interaction patterns exist, such a the official ITU MSC standards MSC-96 [IT96] and MSC 2000. The UML [OMG11b] provides sequence diagrams that with version 2.0 of the standard have incorporated many elements of the MSC 2000 specification. While all these notations provide powerful syntax for the specification of distributed component interactions, only our MSC-dialect provides the precise semantics interpretation suitable for our formal system model.

Harel et al. [DH01] introduced Life Sequence Charts (LSCs) as an extension of MSCs with an emphasis on richer expressiveness, in particular towards the "lifeness" of specifications. They enabled the specification of forbidden behaviors, possible and required behavior and embedded the notation within a methodological framework in the tradition of Harel's Statecharts [HP98]; it also applies a state based semantics. This approach is comparable to the MSC-based interaction specification technique we apply, which also provides rich expressiveness, modular decomposition and a variety of control constructs and operators. Our approach, however, offers a semantic interpretation based on FOCUS and in particular offers the well defined "join" operator to combine separate specifications.

The approach we approach for the specification of interaction patterns supports defining Quality-of-Service properties similar to the approach proposed in the UML Profile for Schedulability, Performance, and Time (UPSPT) [OMG03b]. In fact, our service definition supports more general QoS specifications than what is possible in UPSPT, as discussed in [AKMP05], because it enables to specify such properties spanning multiple components.

4.4.9 Data View

The *Data View* specifies details related to the information flow between components. It also specifies how the data state of components is defined. Figure 4.33 shows a projection of the logical architecture model element domain model depicted in Figure 4.8, showing only the entities and associations that are present in the Data View.

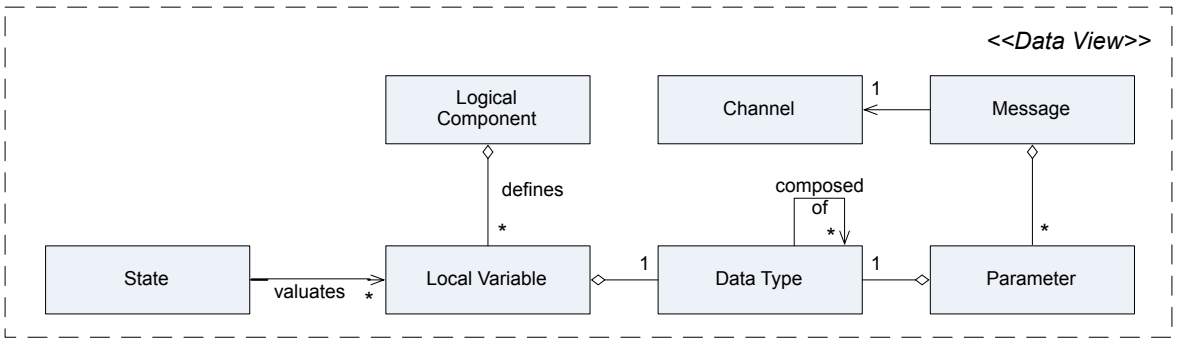

Figure 4.33: Data View domain model

Figure 4.34 shows exemplar data definitions for the BART system level service "InterlockingDataExchange", as introduced above. The data definition identifies the scope it applies to, here "SYSTEM" and specifies communication messages and data structure objects. The syntax of the definition file is based on the YAML language[11]. A similar language is used in the OOI Cyberinfrastructure and the SciON framework to define service interfaces and object data structures [OOI14, Sci15].

```
Data Definition SYSTEM
 Messages
  InterlockingDataExchange_update_in:
   doc: Used in InterlockingDataExchange service to update
    system with external interlocking system status
   sender_roles: Environment
   receiver_roles: System
   parameters:
    interlocking_data: !InterlockingDataObject

  InterlockingDataExchange_update_out:
   doc: Used in InterlockingDataExchange service to update
    environment with train status
   sender_roles: System
   receiver_roles: Environment
   parameters:
    train_data: !TrainDataObject
```

```
Objects
 IdentificationHeaders:
  communicator: ""  # ID of communicating process
  authentication_info: {}  # Additional security info
  prepare_timestamp: datetime   # Time of preparation
  error_status: 0  # Any error code if any
 InterlockingDataObject:
  id_headers: !IdentificationHeaders
  interlocking_data_segments: []  # List of detailed data
 TrainDataObject:
  id_headers: !IdentificationHeaders
  #@ContentType=TrainDataSegment
  train_data_segments: []  # List of detailed data
 TrainDataSegment:
  train_id: ""
  last_comms_time: datetime
  current_speed: 0.0
```

Figure 4.34: BART system level data definitions

The example defines message types, identified by their message type names. These names contain the service name, a message label (e.g. "update"), and a direction from the current

[11] `http://www.yaml.org/`

component's point of view (e.g. "in") to be systematic and unique, but in the end can be chosen freely. Each message type definition lists applicable message parameters with a detailed content type definition. Additional information exists for each message type, such as a description, applicable sender roles and applicable receiver roles.

The example in the figure also specifies object data structures. These are typically used to represent data structures sent through messages, but can also be used to represent component state, e.g. in local memory or persistently. Object types are identified by name and can subsequently be nested within other objects using the !ObjectType notation. Objects have attributes, defined by a type and default value. Attribute types include simple types, such as string (denoted by two double quotes), integers, decimal numbers, lists (denoted by square brackets), associative arrays (denoted by curly braces) and nested object structures as explained above. Objects and their attributes can have comments (starting with the"#" character) and additional decorators, such as to indicate the content type for collections.

4.4.10 Structure View

The BART case example is a complex system with software, hardware, electrical and mechanical elements with intricate interdependencies. It is a highly distributed system consisting of the station control systems, the trains, the interlocking system controlling switches and gates, and a communications subsystem. Within the environment of the system lie responsibilities for manual operation of the system, train scheduling and many others. We will show how this informal understanding informs the decomposition of the system into components, as specified in the *Structure View*.

Constituents of the Structure View Figure 4.35 shows a projection of the logical architecture model element domain model depicted in Figure 4.8, showing only the entities and associations that are present in the Structure View. The main entity is the logical component. The Structure View mainly defines the decomposition of a component into subcomponents and their communication links through communication channels. Some communication links exist with the component's environment through identified interfaces, as identified in the Interface View. These communication links may be with another component of the system, an external system or a user in the environment. This is irrelevant from the point of view of the component.

A component's Structure View must be consistent with the Interface View, which provides detailed identification and specification for communication channels with the environment. This includes the syntactic and behavioral interfaces.

Component Definition We use a *Component Decomposition Diagram* (CDD) to define the inner structure and the external interfaces of a component. A CDD contains a label **cdd**

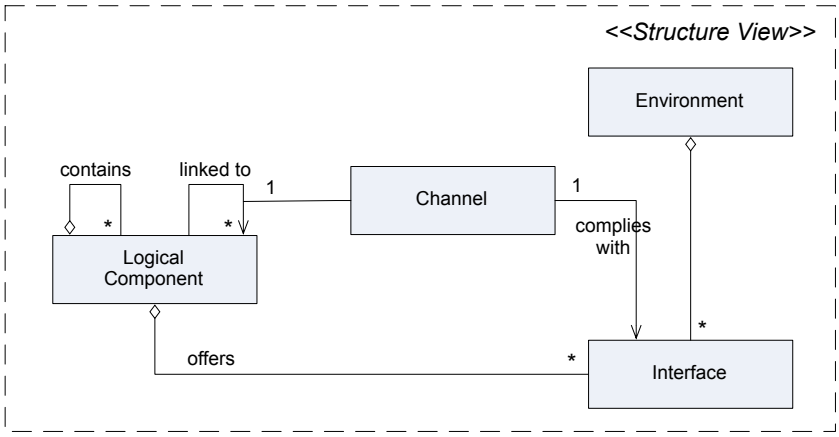

Figure 4.35: Structure View domain model

identifying it as a component decomposition diagram, a unique name identifying the logical component within the system, and a depiction of subcomponents, their communication channels and external interfaces of the component. We use a notation derived from UML2 class diagrams [OMG11b] for our CDDs, similar to the diagrams used in the Interface View. Figure 4.36 shows the decomposition of the SYSTEM (component) of the BART example into subcomponents. On the right it shows the component definition in *ServDL*.

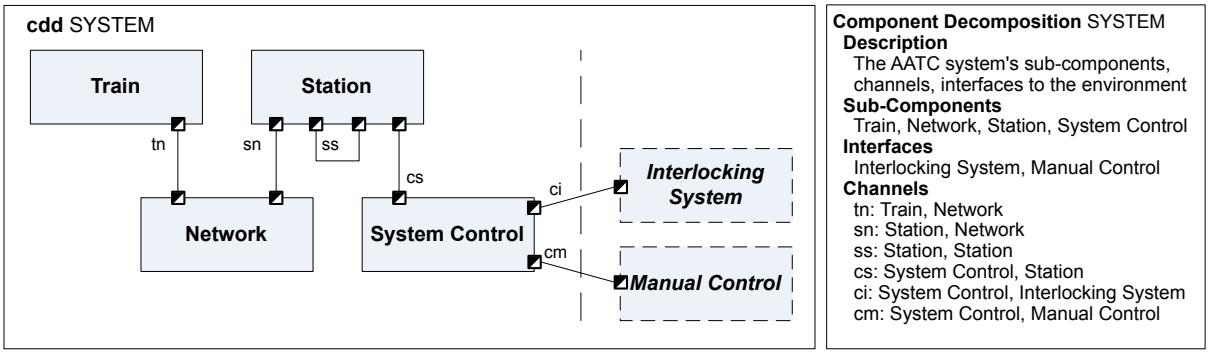

Figure 4.36: BART system component decomposition diagram and *ServDL*

The system described by the requirements is the AATC, which is a part of the overall BART system. We identify it as top level component named SYSTEM. It is a complex system consisting of hardware, software, documentation, organizational parts and supporting systems for maintenance, development etc. Our SYSTEM defines the following subcomponents:

- "System Control", the system controller component (software)
- "Station", the components controlling stating computers (hardware, software)

- "Network", the radio communications infrastructure (hardware, software)
- "Train", the on-board controller software for trains (software update to existing hardware)

4.4.11 Component Behavior View

The Component Behavior View specifies the interaction behavior of a component comprehensively. It is the result of all specifications made in the other model views of a component and provides the basis for subsequent implementation activities. Figure 4.37 shows a projection of the logical architecture model element domain model depicted in Figure 4.8, showing only the entities and associations that are present in the *Component Behavior View*. The logical component interacts with the environment compliant with the specified interfaces. The diagram adds one model element: a "Total Behavior Specification" associated with the component.

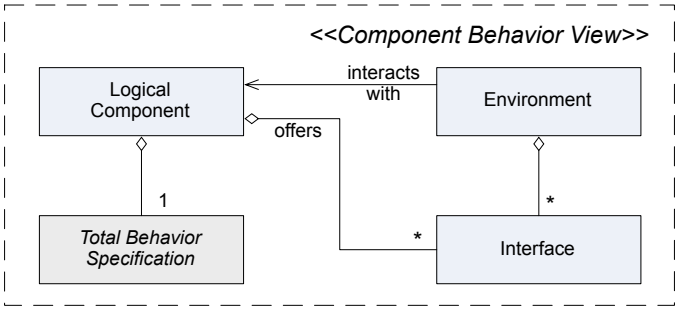

Figure 4.37: Component Behavior View domain model

The Component Behavior View provides a total component interaction behavior specification for a component; this means it provides complete behavior definitions for all possible inputs, covers all exceptional conditions and thereby provides error handling behavior. This view is useful for transitioning to a component implementation. Based on this view, a prototypical component implementation can be automatically generated from the design model. Such an implementation can be integrated automatically into a prototype of the entire system, when combined with component assembly and configuration information as specified in technical architecture views.

Derivation of Total Component Behavior It is possible to derive total component behavior specifications for this model view from service interaction specifications. We introduced a synthesis algorithm based on our formal system model in Section 3.3.5. To validate its feasibility, we supported the implementation of this approach in the M2Code tool described in [AKMP05], applying it to the Central Locking System of a modern automotive control system.

Similar component state machine synthesis algorithms exist, such as the one described in [BFH+07]. There, we showed how to derive component implementations from partial service models, defined via interaction patterns. Figure 4.38, for instance, shows part of a generated state machine for the "CashDesk" component of the CoCoME case study.

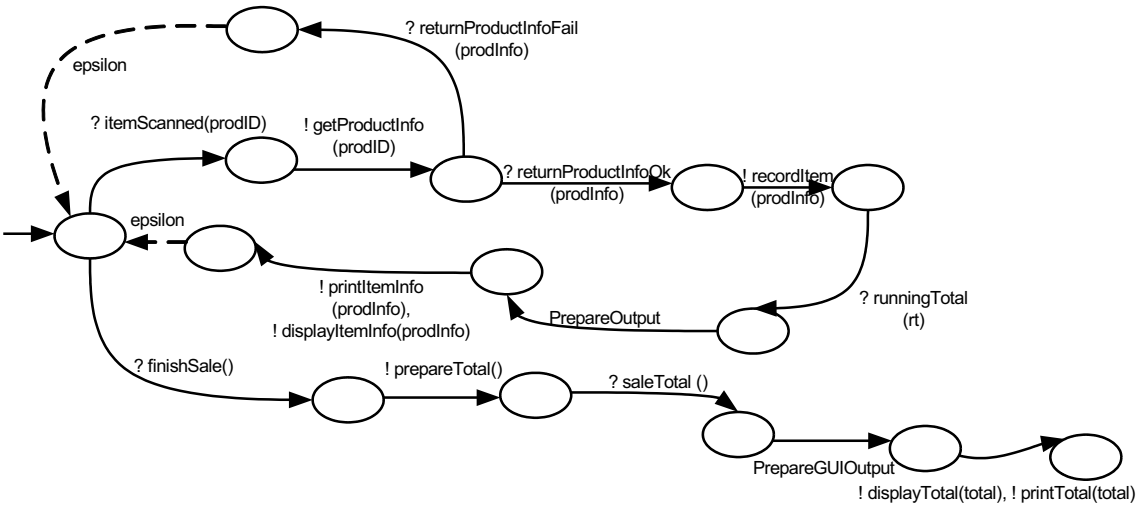

Figure 4.38: Generated component state model for CoCoME "CashDesk"

The actual procedure to derive a total component behavior specification from a number of service interaction specifications involves multiple systematic steps. The first step applies a closed world assumption, meaning that all available interaction specifications are treated as the complete set of component behavior. Prior to this step, new interaction specifications could be added to the Interaction View, as long as they were consistent with the other views of the component, making use of the property that interaction specifications are partial and only constrain the behavior of the component with respect to the specified message exchange sequences. Subsequent steps include translating interaction specifications into state machine notation, synthesizing composite behavior state machines from multiple concurrent interaction specifications for instance as resulting from the projection of multiple roles to one component, and removing redundant and undesirable automaton transitions.

The final result of the synthesis procedure can be complex and unwieldy, depending on the complexity of the interaction specifications, and should not be analyzed or modified directly. Tools need to be applied to further process the state machines, e.g. for generating implementations and for model-checking purposes. We described such tools in [AKMP05].

Related Work Numerous approaches exist for authoring specifications based on state machines [HMU01], e.g. Harel's Statecharts [HP98] and the approach described in [Sch98]. A formal FOCUS based semantics for state-transition diagrams (STD) was introduced

in [GKRB96]. Such specifications can for instance be verified using model-checking techniques [CES83, Hol03] and theorem proving. They can also be used to derive production code automatically.

Our approach is in the tradition of such state machine based specifications. We believe that directly authoring state machine based component specifications is at a too low level to be effective and scalable to larger sized distributed systems, as introduced in the motivation. Directly authoring component specifications requires comprehensive knowledge of a component's functionality and necessitates early design decisions about component interfaces and interaction dependencies. Such specifications can be very detailed, prone to error, difficult to maintain and very inflexible to refinement and change. Instead, our approach focuses on providing more abstract specification techniques based on roles, modes and interaction patterns and provides methodological support for mapping such partial behavior specifications to a component decomposition and automatically generating comprehensive component state specifications from these mappings.

4.5 Benefits of the Formal Model

Our process artifact model was carefully designed to be able to represent the concepts of our formal model, unfolding its semantic space into an engineering methodology. A well-defined artifact model based on a comprehensive formal model offers substantial benefits for the development of distributed systems. The following list shows some of these benefits by design model view:

- *Interface View:* A component's interface can be specified precisely, e.g. supported by an authoring tool, and the meaning of syntactic and semantic definitions can be reasoned on. Can check whether an interface is compatible with another interface, e.g. another component or an external system; can check whether all interfaces of a component's subcomponents refine the component's interface.
- *Service View:* Can check that all services are a refinement of the component interface; can check that a service interaction specification is causal.
- *Modal View:* Can determine all mode combinations for a component with all parent components; can check whether modes are consistently referenced in role and interaction views.
- *Role View:* Can check that roles reference valid operational modes and that operational model transitions are valid; can check that roles map to valid logical subcomponents; can check that all roles have a component mapping.
- *Data View:* Can check whether the interface is consistent with provided data definitions; can check whether service and interaction views are consistent with data definitions.

- *Interaction View:* The interplay between multiple roles can be specified precisely, e.g. supported by an authoring tool, and any invalid operators, message types and role references can be detected. Can check whether an interaction specification is a refinement of a service specification, and is consistent with the component interface. Can check whether different interaction specifications are consistent and can be combined. Enables successive refinement of interaction specifications with messages, referenced interactions, local actions, state labels and timing information. Can check whether there is non-determinism and if interactions are implementable; can automatically generate a state machine from one or multiple interaction specifications.
- *Structure View:* Can check whether communication channels are consistent with interactions among roles and reflect data definitions; can check whether component interfaces are compatible for linked components; can refactor a component into multiple components.
- *Component Behavior View:* Can check that component behavior is a refinement of the component interface; can model check component behavior against properties of a formal specification.

These benefits can be exploited in development tools, thereby directly increasing development efficiency when compared to less formal development approaches. In particular, the immediate detection of specification errors and inconsistencies among different views leads to higher productivity, see also Section 6.2.2. The ability to formally verify specifications and to automatically generate code and prototype implementations makes quick development cycles possible and enhances the correctness of specifications. In the outlook in Chapter 7, we will show how formal models for requirements and service deployment promise to provide further benefits, for instance detecting unwanted feature interactions and structuring a system's usage functionality.

4.6 Summary

In this chapter, we have introduced the artifact model for our service-oriented development process. We have explained possible structures of such an artifact model and decided to lay out artifacts according to the main system engineering activity that produces them. We have introduced domain models identifying the artifacts and their relations; in particular, we have provided a comprehensive domain model for the artifacts related to the views for the logical architecture design model. We could see a correlation of development artifacts to the abstraction levels relevant to the development of service-oriented systems.

We have introduced 8 distinct, closely related views for the logical architecture design model. These include the Interface View as the defining "black-box" specification for a system or a component, high-level views refining component structure (Role View and

Structure View) and behavior (Modal View, Service View, Interaction View), and comprehensive views defining data and component behavior. Methodology exists linking these views, to be explored further in the next chapter. Consistency rules constrain possible designs and refinements. The system design model supports a hierarchical system decomposition. The decomposition of behavior and structure on one level relates to corresponding detail views for the decomposed model elements, such as child components. Depending on the design strategy applied, this supports top-down refinement of the system into components, and bottom-up composition of the system out of smaller components, and any combination thereof.

The well-defined artifact model introduced in this chapter enables us to subsequently introduce a development process connecting all the views and applying methodology. We will show this in the following chapter.

5 A Comprehensive Service-Oriented Development Process

In this chapter, we complete the definition of our service-oriented development process. We define activities and role responsibilities leveraging the artifact model introduced in the previous chapter. We emphasize the iterative nature of the development process and explain in more detail some critical activities such as iterative refinement. The result is a comprehensive development process applicable to distributed reactive systems. Subsequently, we show how to embed our development process into an existing system engineering process framework, the V-Modell XT. Thereby, our process becomes readily applicable, in particular to organizations already applying the V-Modell.

Contents

5.1 A Core Service-Oriented Development Process

In this section, we introduce the core of our service-oriented development process, complementing the process artifact model introduced in the previous chapter with a process activity model. We begin by laying out the guiding principles we followed when defining this process and in particular its activity definitions.

5.1.1 Guiding Principles For Process Activity Definitions

A development process contributes structure and organization to the application of system development methods and tools. As discussed in Section 2.4, development processes strive towards defining systematic activities with clear responsibilities for the modification of process artifacts. Activities decompose into work steps. We applied the following guidelines when defining the activities and work steps of our process:

- *Simplicity:* A work step of an activity can be completed by one individual within a limited amount of time. It must be possible for the individual to understand the purpose, needs and consequences of the work step within the broader context of its activity and the entire project. If too complex, a work step must be broken down into smaller work steps.
- *Limited scope:* A work step only results in artifact modifications of limited scope, such as the requirements for a certain subsystem, the design of a specific component interface or the implementation of a software module. It must be clear when a work step or an activity is completed.
- *Defined outcome:* The activity description clearly states the artifacts modified by its work steps as well as the nature and extent of the change. Quality assessors must be able to evaluate the correct completion of an activity by examining the modified work products with respect to form, content, and consistency with dependent artifacts.
- *Clear responsibility:* The activity definition assigns clear responsibility for the completion of the activity to project roles. Support roles may be named as well. Project roles must be defined in a way so that they can be assigned to individuals in a project.
- *Method and tool guidance:* The process suggests applicable methods and tools, thereby assisting individuals who carry out the work. Goal is to maximize effort spent on the work result, and minimize the time spent on finding and understanding possibly useful methods and tools.
- *Support parallel work:* The number of activities required to be completed before other activities can start is restricted to a minimum. The process enables multiple individuals to work in parallel on related artifacts and offers ways to reconcile any differences in an orderly manner. The process suggests a preferred work order and

minimizes the cost of parallel work on not yet baselined work products, for instance by modularization and clear product consistency rules.

- *Enable iterations:* The process supports the modification of any artifact at any stage in the project—in a controlled way consistent with other artifacts—in order to remediate inconsistencies, omissions and errors. The process strives to minimize the cost of such corrections and never bases the success of a subsequent activity merely on the correctness of a preceding activity.

5.1.2 Process Overview

Our process applies service-oriented concepts throughout all development activities from requirements engineering to implementation. It suggests suitable design models, modeling notations and supporting methodology. In this section, we provide a high-level introduction to our service-oriented development process. This process is based on the artifacts and model elements defined in Section 4.3.

Artifacts across abstraction levels Figure 5.1 illustrates important development artifacts and model elements. It arranges the artifacts according to their abstraction levels. As we will see in subsequent sections, these abstraction levels correlate with the main development activities. Important process activities need to be performed when transitioning between the model artifacts and abstraction levels. See [BFH+07] for a detailed discussion of the model elements for the architecture and design levels using the CoCoME case study.

Figure 5.1 illustrates essential dependencies between model elements and provides indications for how to reflect them systematically. Arrows refer to activities and steps in a process with a sound methodological foundation; labels next to arrows indicate the kind of activity. For instance, creating a Logical Component Model based on a Service Model requires the application of *transformation and refinement* operations. While doing so, a realization of a higher-level specification—the service model—gets developed. In the previous chapter, we showed that this can be accomplished by generating a Logical Component Behavior View from other views associated with the same component. The underlying theory—formally expressed in Section 3.3.7—provides the necessary *Semantic Interpretation* that enables the use of a component synthesis algorithm. The aggregate set of all component behaviors—representing the designed system—can be *verified* against the Formal Specification. The formal model provides the foundation for such verification activities and supports their automation. The implemented system can be *tested* against the requirements.

The development process defines activities and work steps, applying clearly defined operations to process artifacts, such as the ones named in the figure. It also provides an ordering of activities, thereby enabling systematic, iterative development, following the guiding principles listed above.

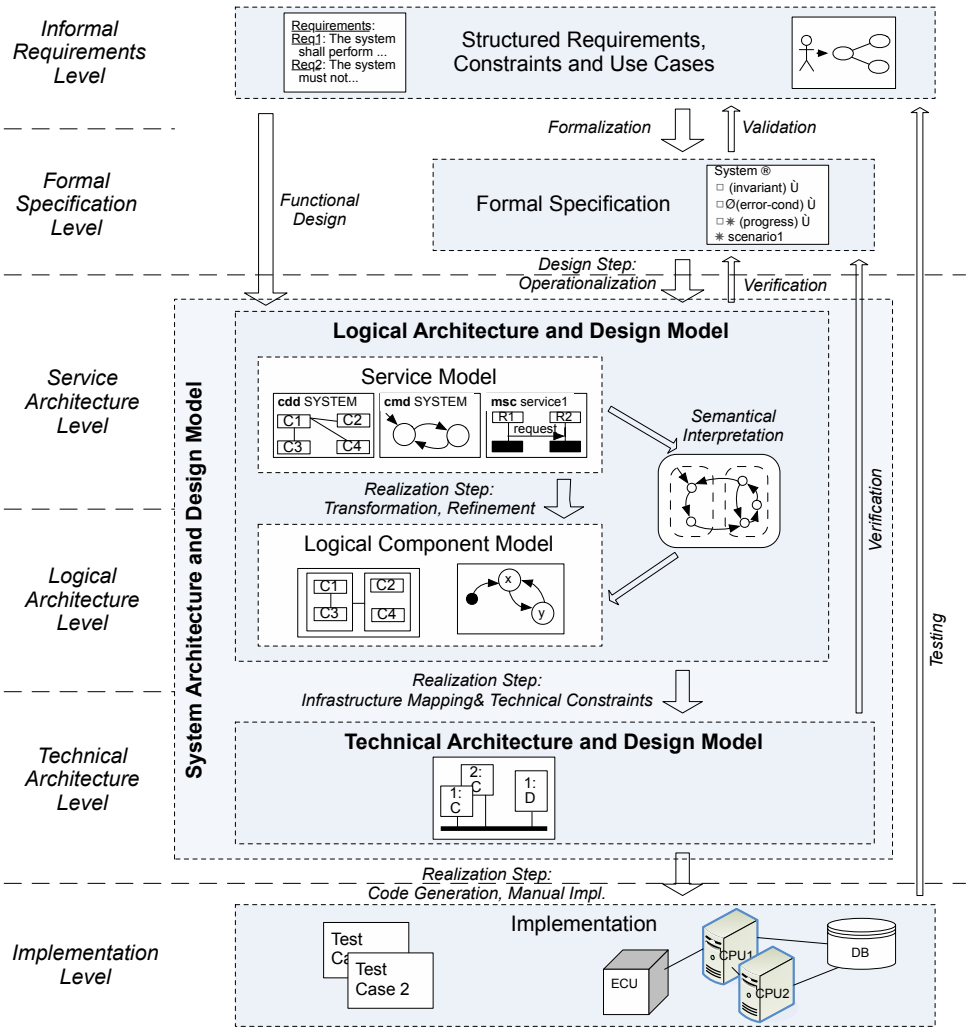

Figure 5.1: System development artifacts grouped by abstraction level

High-level processes At the high-level, our development process is an iterative application of four high-level development processes: *Requirements Engineering*, *Logical Architecture Design*, *Technical Architecture Design*, and *Implementation and Integration*. Figure 5.2 illustrates this high-level view. These processes—sometimes called high-level activities or activity groups—are somewhat aligned with the main development activities introduced in Section 2.2.

In the ideal world and project, these high-level activities would ideally be performed in sequence, in order to maximize benefits for moving from an abstract problem space to a concrete solution space in a single pass, while performing straightforward refinement and dependency tracing between artifacts. Because a single sequential execution of the high-level development activities is unrealistic for most larger sized development projects, the process can be applied iteratively and supports parallel execution of the activities.

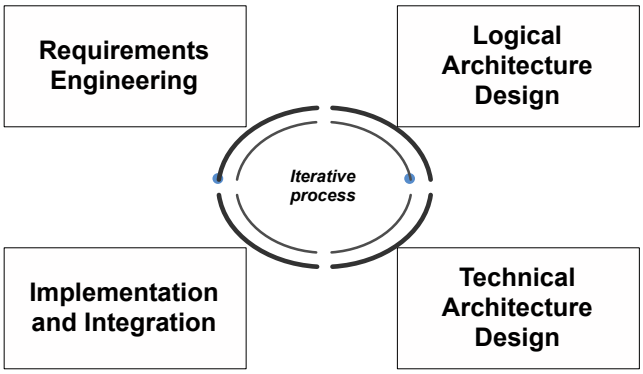

Figure 5.2: High-level view of the service-oriented development process

Subsequent activities feed back into prior activities and the development cycle can be repeated as often as necessary, performing modifications and refinements to the previously developed artifacts, maintaining their consistency. This iterative development process is in the tradition of Boehm's spiral model [Boe88].

In the following, we provide a brief walk through the high-level activities and highlight their main artifacts, activities and role responsibilities. For important activities, we provide detailed activity diagrams and explanations. For the remaining process elements, we refer to Section 4.3, where the core outcomes of our system development approach were defined. We assume that activities describe the detailed authoring of these artifacts.

5.1.3 Requirements Engineering

The *Requirements Engineering* high-level activity, illustrated in Figure 5.3, targets the development of viable requirements for a system under development. This activity performs background research, situational analysis, use case development, requirements elicitation and, if needed, formal system specification. It develops the artifacts described in Section 4.3.1.

Requirements Engineering results in the definition of relevant use cases and their relationships. Use case dependencies can be captured in the form of use case diagrams. Gathered knowledge of the system and its environment can be captured in form of domain models. Figure 5.3 illustrates information flows and dependencies between artifacts, providing a first glimpse at process activities, including, but not limited to: "Identify collaborating system internal and external entities", "Document system use cases", "Identify functional system behavior", "Specify crosscutting quality properties"

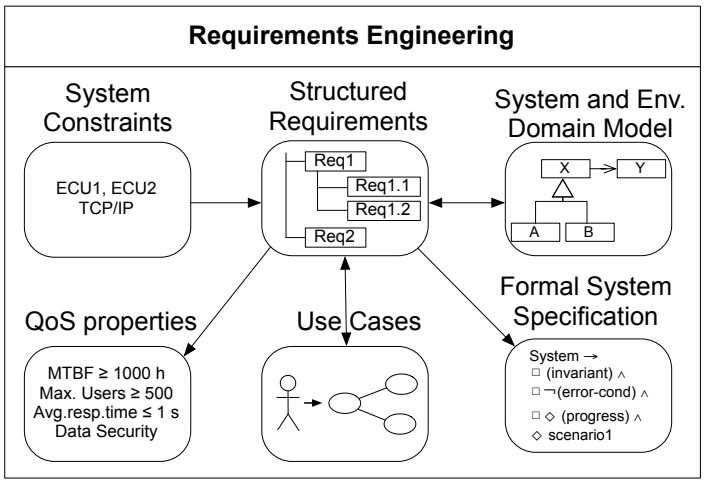

Figure 5.3: Requirements engineering in the service-oriented development process

5.1.4 Logical Architecture Design

The *Logical Architecture Design* high-level activity, illustrated in Figure 5.4 supports the development of the logical architecture and design model based on requirements, use cases and system domain models. The logical architecture model defines the hierarchical system architecture, component interfaces, services, roles, modes and data definitions. It also specifies service behavior using interaction patterns. We introduced all relevant artifacts and model views in Section 4.3.2 and showed essential dependencies between the logical architecture model views, cf. Figure 4.7.

In particular, the decomposition of the system into components supports a modular application of design model views for each component. Some views define the relationship between components and their interplay. Other views specify internal details about components. Process activities and work steps describe the specification of component structure and behavior within model views. They also describe how to relate across model views, keeping the information consistent. Model consistency is additionally supported by the underlying formal system model and can for instance be assisted by tools.

Figure 5.5 illustrates the main activities of logical architecture design in form of an activity diagram. These activities include "Define system architecture" for developing and refining a high-level decomposition of the system into a hierarchical structure and specifying crosscutting concerns. The remaining activities are scoped to an individual component. The activity called "Define component interfaces" results in Interface, Data and Service View specifications for a component. This is the external representation of a component. The internal specification gets developed in "Define component service model" resulting in Role, Modal, Structure and Interaction View specifications. Here a component's substructure, operational models, roles and interaction patterns get developed. Data View definitions apply to both external representation and internal specification. The activ-

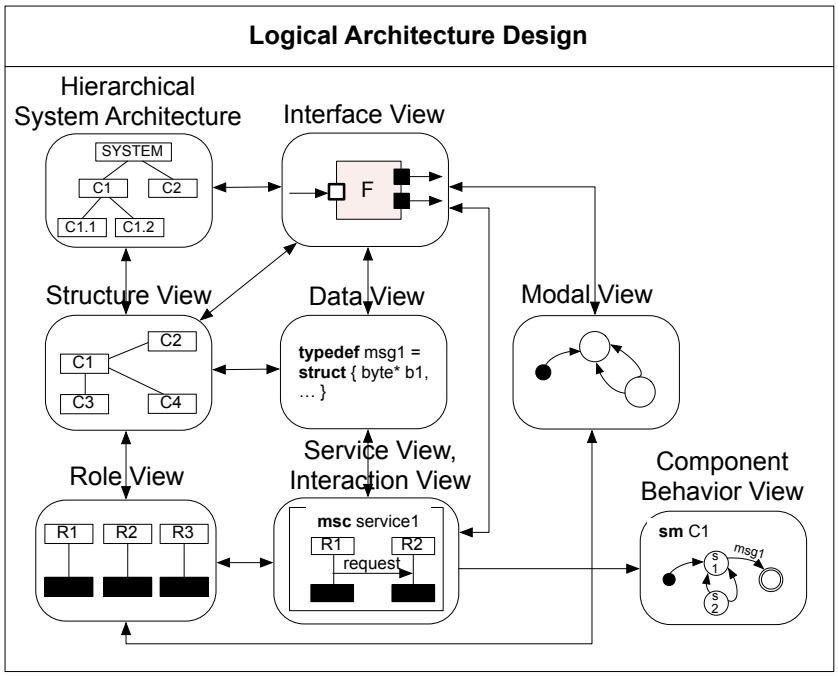

Figure 5.4: Logical architecture design in the service-oriented development process

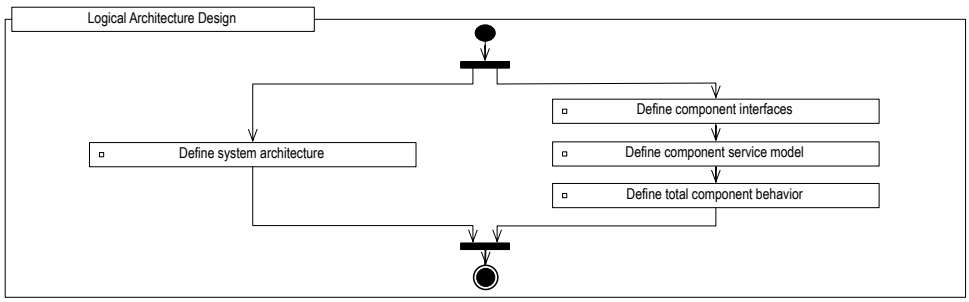

Figure 5.5: Activities for "Logical Architecture Design"

ity "Define total component behavior" results in the comprehensive Component Behavior View. Component behavior can be interpreted as the consistent combination of all of the other component's specifications.

5.1.5 Technical Architecture Design

The *Technical Architecture Design* high-level activity, illustrated in Figure 5.6 develops detailed specifications to be utilized during system integration activities. The logical design model is refined into a valid and viable deployable design, taking into account physical infrastructure and existing technical constraints. We introduced all artifacts and model

views in Section 4.3.3.

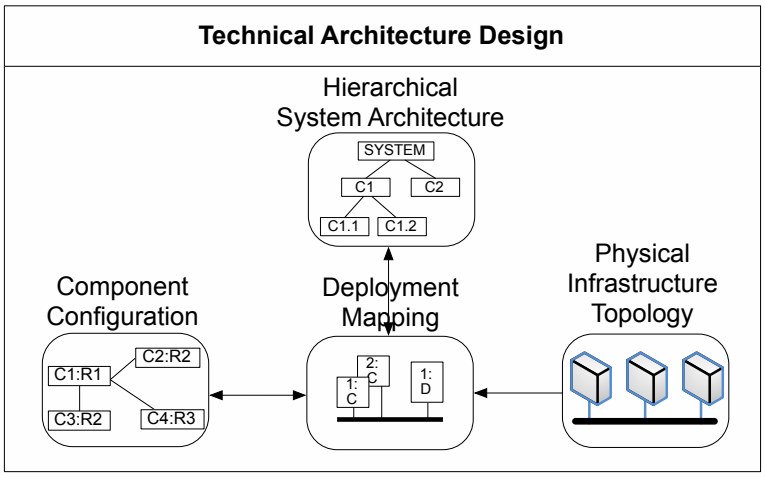

Figure 5.6: Technical architecture design in the service-oriented development process

Important activities include "Identify viable deployment architecture candidates" to document and evaluate variants in deployment configurations, "Assess performance and quality characteristics of architecture candidate" to analyze important properties of a variant, "Characterize the physical deployment environment" to describe specifics of the target system environment, and "Select deployment architecture candidate" to actually decide on a solution in a systematic, documented way. Architectural candidates are descriptions of potential solutions satisfying the requirements and other system constraints, using specific technologies, communication infrastructure, deployment mappings and configurations. Choosing the candidate that satisfies given constraints most optimally for the project is a challenge that the process needs to systematically address. It is important to document the steps that lead to the decision, such that project stakeholders and future users can review them.

Additional specification activities include "Model component assembly", and "Model component configuration" to specify result of the integration of components into larger assemblies and their connectivity, and "Map component configuration to physical infrastructure". Analytic activities include "Verify consistency of deployed system design" and "Check compliance of deployed system design against logical model".

5.1.6 Implementation and Integration

The *Implementation and Integration* high-level activity, depicted in Figure 5.7 takes the blueprints defined in the design activities and implements respectively integrates components, assemblies and the entire system. Artifacts range from source code to executables,

deployable packages and the implemented system. We introduced the relevant artifacts in Section 4.3.4.

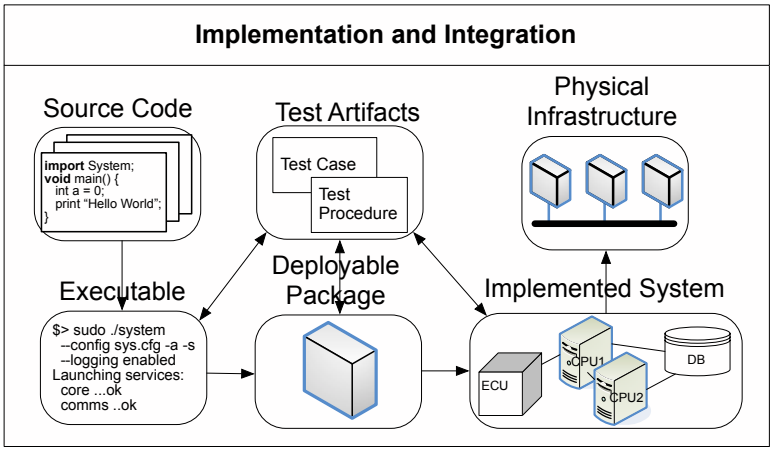

Figure 5.7: Implementation and integration in the service-oriented development process

Important construction activities include "Implement component behavior and interface", "Assemble components", "Build executable" and "Create deployable package". These activities support the steps of building the pieces of the system and integrating them into larger assembles and the entire system. This covers the development of source code using Integrated Development Environments (IDEs), the application of unit testing by the developer, developing hardware units, and using build systems to package, build and test executables. It also covers subsequent steps of combining executables into deployable packages and the implemented system.

Build and deployment activities include "Setup build environment" in order to prepare for the production of software and hardware components and "Deploy and install deployable package" in order to bring a ready package into its target environment, applying installation procedures and configuring it for operations.

Verification support activities include "Develop test case and procedure" and "Derive test cases from logical and technical design model" to develop the actual tests and test procedures. "Setup test environment" prepares for the environment for the actual testing process and may range from setting up a hardware test lab to an automated software test system. "Test executable or deployable package", "Test implemented system as installed", and "Test implemented system as deployed and operational" are verification activities that compare the constructed artifacts against their specifications. The activity "Validate that the correct system was built" requires process stakeholders and end users to evaluate the system against expectations given how the system performs.

Documentation activities include "Develop operating procedures and ancillary documentation" to develop the reference materials for users and operators. Many other supporting

activities and work steps exist developing the supporting materials required for the operation of the system, often referred to as logistics support.

5.2 Service-Oriented Development Activities

In this section, we describe in more detail specific activities important for the iterative development of distributed reactive systems. In particular, we describe selected activities and work steps to specify a logical architecture design model, and to systematically refine it following the principle of compositionality. We also show how to iteratively modify and refactor design models. We reference the process artifact model introduced in the previous chapter. Where useful and not redundant with the artifact model, we show examples for model views and artifacts from our BART running example. We also describe how our formal model contributes to facilitating the correct execution of the activities and work steps, and the assessment of outcomes.

5.2.1 Activity "Define component interfaces"

This activity results in a consistent component interface specification that is traced to system requirements and assessed for consistency with related specifications. The activity defines several work steps; Figure 5.8 illustrates the flow of work steps in form of an activity diagram.

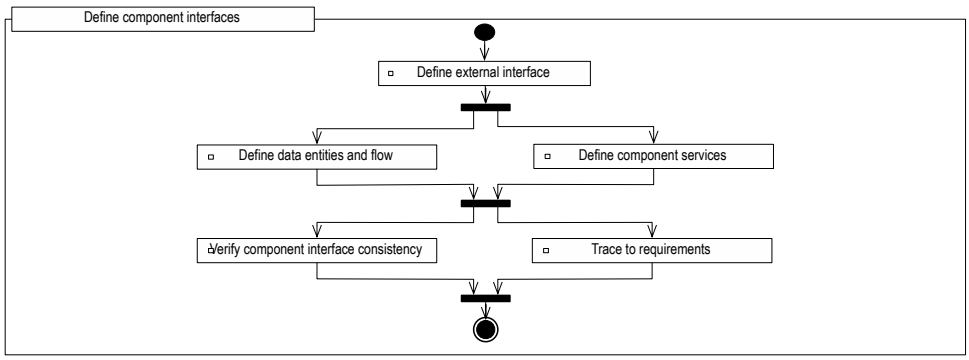

Figure 5.8: Activity diagram for the "Define component interfaces" activity

The initial work step "Define external interface" derives and defines the external interface specification of a component, realized within the design model's Interface View. This work step is followed by the work steps "Define data entities and flow" to refine the Data View and "Define component services" to refine the Service View. These two work steps can be executed in parallel. After their completion, the logical design model provides the external interface specification of a component.

In order to guarantee consistency of the component's interface specification with any superior component, the system, or other artifacts, the process provides the work step "Verify component interface consistency". Our formal model underlying the process artifact model provides the basis for an array of consistency checks. The component's external interface can be checked against the parent's specification with respect to valid refinement, such as property and interface refinement, cf. Section 3.3.6. These and other checks detect issues such as mismatching component identifiers, unsatisfied component interactions, misaligned data types of communication messages, and general problems with the formal correctness of the interface definition. The parallel work step "Trace to requirements" creates any linkages from requirements to elements of the system design. These linkages will be contained within the requirement artifacts and should ideally be managed using tool support, such as IBM Rational DOORS and AutoRAID [SFGP05].

5.2.2 Activity "Define component service model"

This activity results in the elaboration of a component's logical design model, resulting in a comprehensive specification of the component's service model. This model includes the Role View, Modal View, Structure View and Interaction View as described in the previous chapter. Figure 5.9 provides an activity diagram for this activity with the flow of work steps.

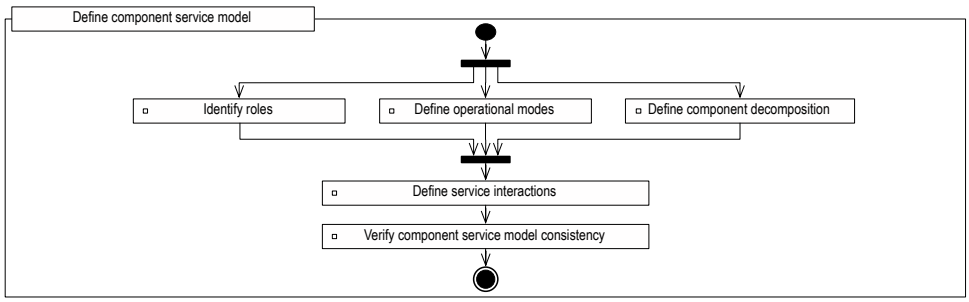

Figure 5.9: Activity diagram for the "Define component service model" activity

Initial work steps that can be executed in parallel include "Identify roles" to specify roles in the Role View, "Define operational modes" to list modes and model transitions in the Modal View, and "Define component decomposition" to decompose a component further into subcomponents. In the following sections, we will show how these work steps can be broken down even further to support systematic iterative revision of the design model.

The subsequent work step "Define service interactions" provides the designer with the opportunity to define component service behavior in the form of interactions. Interactions can be specified, for instance, using the MSC dialect introduced in Section 3.3.2. High-level MSC diagrams provide a means to structure interactions into modular specifications

that then can be realized in form of MSCs diagrams. Every service needs to be defined using such MSC diagrams. The MSC notation has a formal semantics providing a precise syntactic and semantic interpretation for subsequent code generation and verification steps.

The activity concludes again with an analytic work step "Verify component service model consistency" to apply any consistency rules of the service model elements to the components interface specification and any other related specifications. This work step can compensate for any inconsistency introduced by parallel execution of design activities for multiple components or by parallel execution of work steps for one component. Issues within a MSC specification, such as non-local choice, nondeterminism, lack of realizability, messages inconsistent with the service interface, invalid use of roles, and inconsistencies across MSCs can be detected easily, enabling the designer to quickly correct the model. Figure 5.10 shows an example consistency error detected by the SODA tool, after analyzing the underlying design model and any cross-references. In this case, model elements were not referenced within a MSC specification. See Section 6.2.2 for more details. The application of artifact consistency rules as well as the assessment of model views and artifacts against formal artifact criteria provide the necessary quality control to complete the manually authored views of the logical architecture design model artifact.

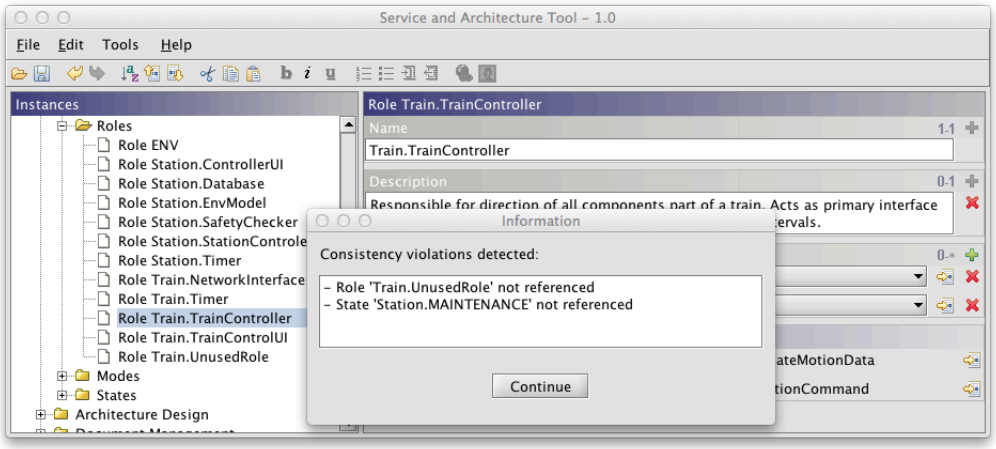

Figure 5.10: Executing consistency checks in the SODA tool

5.2.3 Activity "Define component decomposition"

Figure 5.11 provides an activity diagram for defining a component decomposition. This activity provides work steps for refining higher level elements of the design model structurally, as well as for modifying the design model.

The work step "Add subcomponent", for instance, enables a designer to define subcomponents for a given component, while adhering to the mode, data and interface specifications

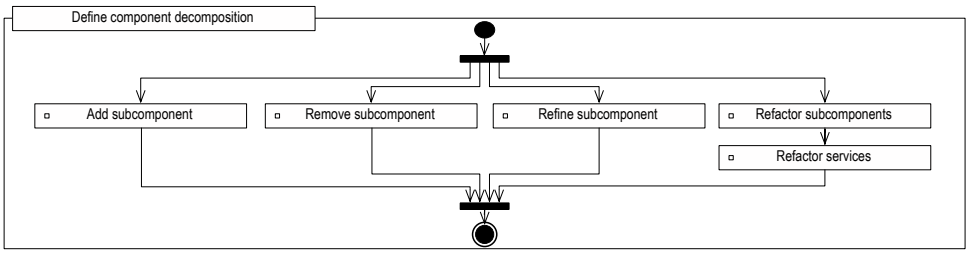

Figure 5.11: Activity diagram for "Define component decomposition" activity

of the component that is getting refined. Correct refinement with respect to the formal model can be checked automatically and when needed. The work step "Refactor subcomponents" enables a designer to apply a systematic model transformation to one or multiple subcomponents of a component. Examples for such refactoring transformations include splitting one subcomponent into two, merging two subcomponents into one, and changing the name of a subcomponent. More complicated refactoring transformations exist as well. The refactoring step needs to be followed by a matching transformation step "Refactor Services" for the component's service model view. We will show model refinement and refactoring strategies in more detail below.

Other work steps exist to modify the design model in a controlled way, such as "Remove subcomponent". These work steps have a defined scope and can be performed somewhat in isolation, optimally supported by an automated tool adhering to the formal model. The work steps, however, have the potential to leave the design model temporarily inconsistent. It is up to the designer to apply compensating work steps on other model elements in order to satisfy the design model product dependencies. For instance, the removal of a subcomponent requires the removal from all model views where it is appearing, potentially cascading to further model views. Such operations are best executed with tool support.

Figure 5.12 shows the result of a very simple decomposition of the BART Train controller component into multiple "Car" components. On this level, the Car components expose interfaces to "OnBoard Operators" as part of the system environment.

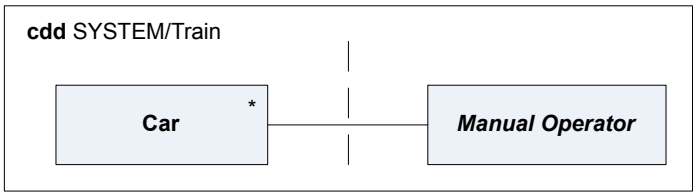

Figure 5.12: BART Train component decomposition

5.2.4 Activity "Define operational modes"

Figure 5.13 provides an activity diagram for the activity of modeling operational models and mode transitions. This activity provides work steps for refining higher level elements of the design model behaviorally, as well as for modifying the design model.

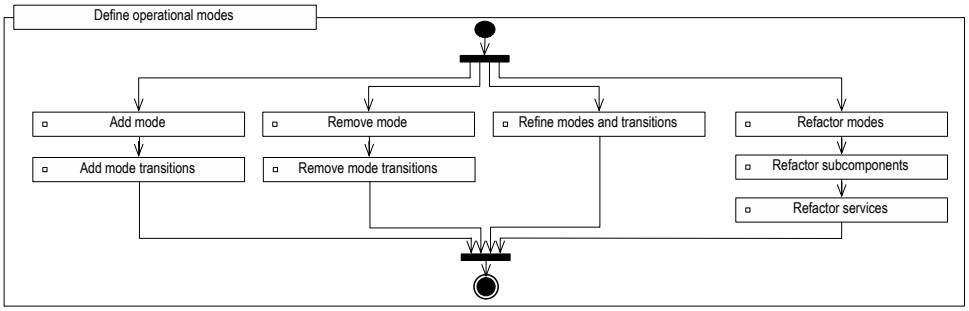

Figure 5.13: Activity diagram for the "Define operational modes" activity

Similar to the component decomposition work steps described above, this activity provides refinement work steps, refactoring work steps and other controlled model modification work steps. Defining the modes of a component is directly related to the definition of the component's mode transitions, reflected in the sequence of work steps.

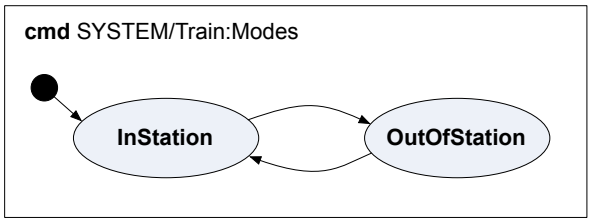

Figure 5.14: BART Train component modes

Figure 4.24 in the previous chapter showed a mode automaton with the system's main operational modes. Figure 5.14 shows a state automaton with two modes of the BART Train controller component. The modes "InStation" and "OutOfStation" are mutually exclusive and can alternate.

5.2.5 Activity "Define total component behavior"

The described activities of specifying and verifying the logical design result in a comprehensive solution if done correctly and consistently. Services, refined into roles, modes and interactions, together with the definition of subcomponents and data elements provide a

comprehensive, albeit still partial specification of a system component. The system designer has the opportunity to iteratively develop the design model and to use analytic tools, verifying model consistency and support of intended properties.

A completed service model for a component can either be refined into subcomponents if too complex or not yet modular enough, or it can be prepared for implementation. The activity "Define total component behavior" provides the work steps to perform the latter. If the implementation, for instance subsequently evaluated by analysis or rapid prototyping, does not match the designer's expectations, the process can be applied iteratively.

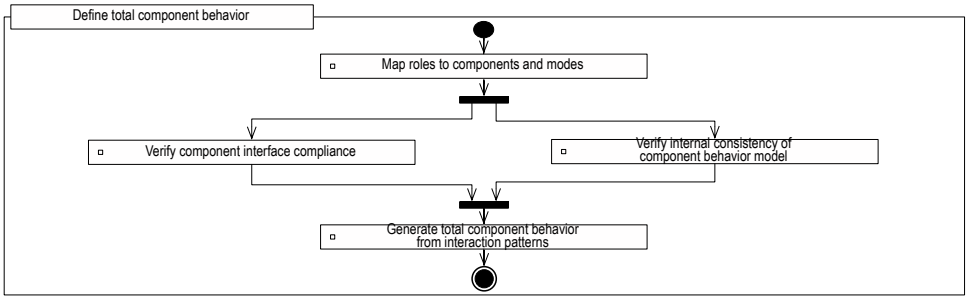

Figure 5.15: Activity diagram for the "Define total component behavior" activity

Figure 5.15 provides an activity diagram for defining total component behavior for a designated parent component. The first work step is the mapping of roles to subcomponents. After this mapping, final consistency analyses ought to occur. Both the component's compliance with specified interfaces and internal component consistency need to be analyzed, ideally supported by tools. Figure 5.10 above shows an example role consistency error detected by a tool.

Once the model appears consistent, the most important work step is to "Generate total component behavior from interaction patterns". Taking all existing independent, partial interaction specifications between roles, combined with their mapping to components leads to complete behavior descriptions of each subcomponent as part of the component behavior views. A set of partial models results in one total model per subcomponent.

[Krü00b] explains several possible interpretations of MSC-based interaction specifications during such a totalization step: *Existential*, where the system may not prohibit the execution of the MSC in all executions. *Universal*, where each execution of the system must show the behavior of the MSC, among other possible behaviors. *Exact*, where the system must exactly show the behavior as expressed by the MSC and no other behavior. Additionally there is a *negative* interpretation of MSCs, where no execution of the system must show the behavior expressed by the MSC.

The behavior synthesis algorithm applied in our approach, cf. Section 3.3.5, applies an exact interpretation. This is a direct benefit of having a strong formal foundation. Given an interaction specification, likely substructured using High-level MSCs into more modular

interaction patterns, it derives a subcomponent's behavior by performing a "closed world assumption". First, it determines all subcomponent occurring as part of the interactions by applying the role mapping. Then, it constructs state automata for all possible subcomponent states given inputs and outputs occurring in interaction patterns. If a subcomponent appears in multiple parallel interactions, the algorithm computes product state automata, eliminates unreachable states and optimizes the resulting automata to practical size.

One of the strengths of our service-oriented approach is that it combines the benefits of existential and universal interpretations of interaction patterns through the application of operational modes. Individual service specifications, as given by subsets of MSCs can be seen as existential in the sense that they apply only for a given set of modes of the parent component. For one mode, however, all service specifications that apply get interpreted universally. This means that all system executions in a particular mode must exhibit all service specifications defined for this mode. No individual service specification shows the exact behavior; they are partial. Only the combination of all parts eventually results in an exact specification.

This exact interpretation, see [Krü00b], does not suit all phases of the system development process. During abstract high-level design and requirements analysis, we use nondeterministic choice (alternatives) to specify existential behavior, i.e. scenarios. We use property refinement, see Section 3.3.6, to iteratively remove nondeterminism.

5.2.6 Activity "Define system architecture"

Developing a logical system architecture and design model implies performing the above listed activities multiple times. The primary driver is the system architecture. Activities are performed dependent on the decomposition of the system into components and subcomponents. Secondary driver are project iterations. Activities need to be performed iteratively, potentially for every component of the system architecture, dependent on the nature of the changes.

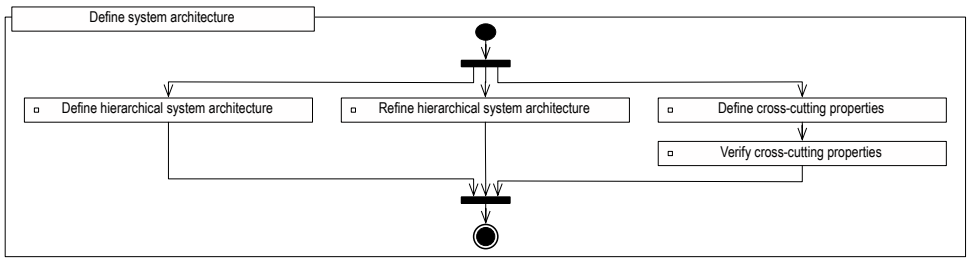

Figure 5.16: Activity diagram for the "Define system architecture" activity

The overall decomposition of the system into components can be depicted in form of a tree. Figure 5.17 shows a component decomposition for the BART system. The system

is the root. It consists of four types of child components, which define child components themselves. Component decomposition and the general definition of the system architecture must be consistent with the formal system model, in particular the notion of rooted hierarchy, see 3.3.3.

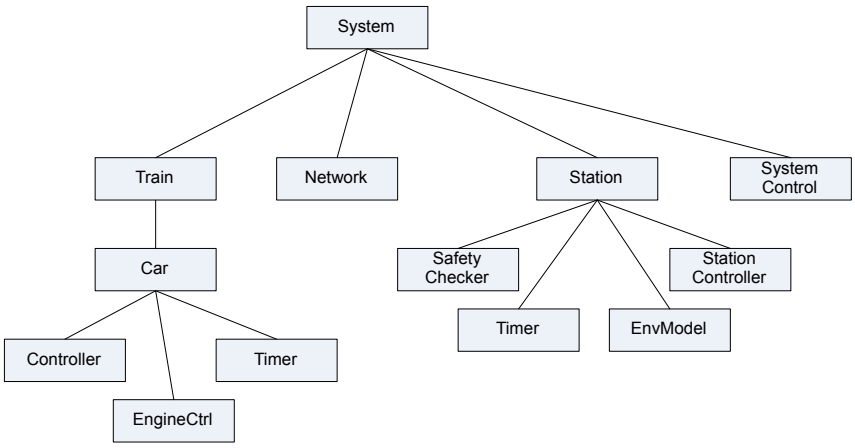

Figure 5.17: Hierarchical system architecture for BART

5.2.7 Modifying the Service-Oriented Design Model

A design model changes heavily during the various phases of system development. If an architecture is hierarchical, designs for subcomponents can be developed somewhat independently. If requirements change or if prototyping eliminates design candidates and requires the change of architecture and design, the design specification needs to be adjusted. Designers typically start with more abstract specifications, such as interfaces and services, and subsequently develop more concrete specifications, such as interaction patterns and component behavior state machines.

Many larger-scale projects lay out their deliverables in increments or iterations, naturally requiring augmented and revised designs. We will discuss iterative elements in the development process in more detail below, see Section 5.4.

Changes to a system's design model may be temporarily inconsistent. Designers may choose to develop a design over the course of days or weeks, temporarily leaving the design model inconsistent across views or with respect to other project artifacts. Towards the end of their modification, they would strive to remove all inconsistencies and publish a new design baseline for subsequent activities to use. A development process and tool support can strongly support such an approach by analyzing a design specification for inconsistencies, by offering automated ways of making defined specification changes, by keeping a log of changes and an ability to go back to prior versions, and by keeping traces

to related artifacts. The development process may additionally mandate analytic activities, in order to verify consistency of artifacts across the project.

Changes to a system's design model may also be scoped to a certain system element. If these changes do not extend beyond the system element, it enables multiple designers to work in parallel without much interference. A hierarchical system architecture with defined, stable interfaces are prerequisites for such an approach.

All these different ways of manipulating a design model can be roughly categorized as:

- *Refinement:* Modifying an artifact in a way that preserves its properties and reduces the amount of uncertainty. Eliminating certain system behaviors is one kind of refinement. Reduction of nondeterminism is a refinement of a design model to bring it closer to an implementation. Refinement also applies when an artifact gets augmented or replaced by consistent artifacts on the next level of hierarchical decomposition, for instance a component decomposed into subcomponents that remain true to all of the component's properties.
- *Refactoring:* Modifying an artifact in a way that preserves its properties by applying defined transformation steps. It is often a special case of refinement, with a high potential of automation by tool support. Examples include splitting a component into two that provide the same behavior and renaming a component consistently within the entire system model.
- *Extension:* Adding system elements to a specification, while maintaining the properties of the existing ones. This can also be a special case of refinement given there are only additions to the design model and no changes to existing specifications.
- *Modification:* All other kinds of modification. Typically requires explicit consistency verification steps to ensure that the modified artifact remains internally consistent and consistent with any dependent artifacts. For instance the deletion of a model element or the change of a service interaction specification.

In the following section, we will explain refinement and refactoring in more detail. Further below, we will cover other ways of modifying the design model in an iterative development process.

5.3 Refinement of Service-Oriented Design Models

In this section, we explain available refinement notions for various elements of the logical architecture design model. Some refinement notions apply to more than one view of the integrated design model.

5.3.1 Refinement of Interaction Specifications

A system's or component's services are defined as interaction patterns, for instance using the MSC notation introduced in Section 3.3.2. Our formal system model enables us to apply a precise semantic interpretation to service interaction patterns. This enables us to apply formally defined refinement operations. The following refinement notions apply to interaction patterns, cf. [Krü00b]:

- *Binding References:* References in interaction pattern specifications are first identified by name without any given specification. As refinement, existing MSCs are bound to the name, limiting system behavior.
- *Property Refinement:* Reducing the set of possible system or component behaviors by strengthening system preconditions and weakening post-conditions. For instance by reducing nondeterminism, eliminating loop alternatives, adding local action labels and adding state labels.
- *Message Refinement:* Insert a sequence of interactions in the place for an individual message.
- *Role Refinement:* Decomposing a role into multiple roles, while maintaining the external interface.
- *Conditional Refinement:* Adding modes to interaction patterns, thereby reducing their applicability to specific modes only. We provided a formal discussion of this refinement notion in Section 3.3.6.

In our BART running example, the components "Train" and "Station" provide a number of services as specified by an HMSC in Figure 5.18. For a "Train", this provides functions such as controlling the operation of brakes and motor (service "Execute motion command"), providing motion status information to the responsible station (service "Communicate Motion Data"), receiving and validating motion control commands from the station (service "Receive Motion Command"), emergency detection in case of missing commands (service "Detect Emergency"), manual train operation (service "Assume Manual Control") and others. All the services of this example are enabled in parallel, by using the parallel operator for HSMCs, referencing the actual services by name using MSC references.

The "Station" component provides functions such as receiving a train into the controlled station area (service "Receive Train"), hand-off of a train to another station (service "Hand-Off Train"), computing motion commands for trains in the area (service "Compute Train Commands"), receiving and analyzing train motion data (service "Receive Train Status"), exchanging information with the interlocking system (service "Interlocking Data Exchange") and several others.

Binding References When beginning the specification of a component's services, it is often straightforward to derive the services from system use cases and requirements. The

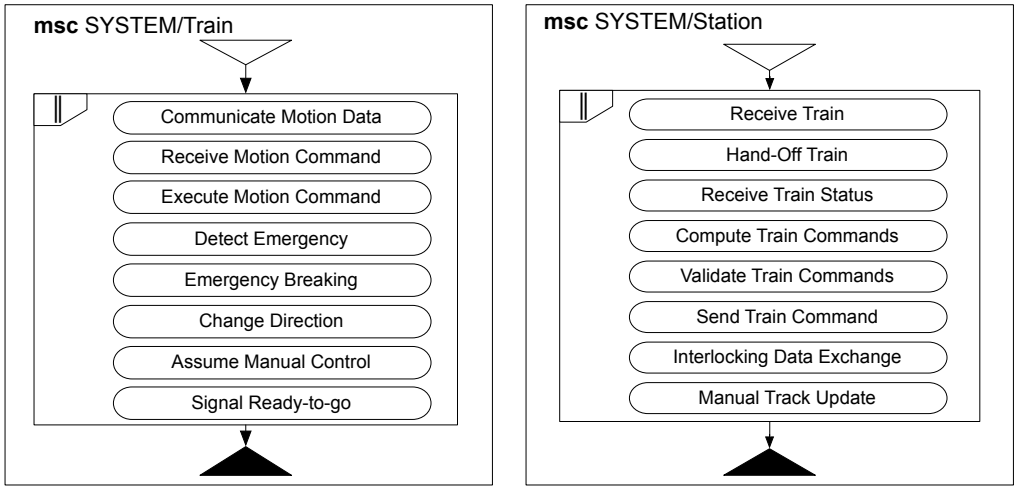

Figure 5.18: BART Train and Station component service specifications

designer should strive to apply good design patterns when designing services, such as keeping them consistently named, modular in their function, combinable with other services, and ready for reuse in different contexts. For instance, services can first be identified by name as shown in Figure 5.18.

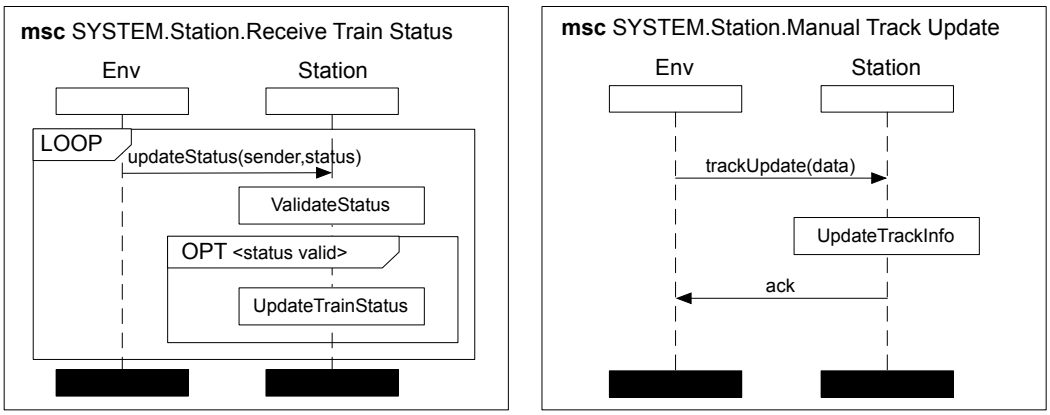

Figure 5.19: BART Train component service interaction specifications

Subsequently, service interfaces can be bound to these names through the Binding References refinement. This is as easy as adding an MSC to the system design specification. Figure 5.19 shows interaction patterns for two services of the Train component as MSC specifications.

Figure 5.20 shows interaction patterns for two services of the Station component as MSC specifications. In all service specifications, the environment is represented as role "Env", interacting with a role representing the component.

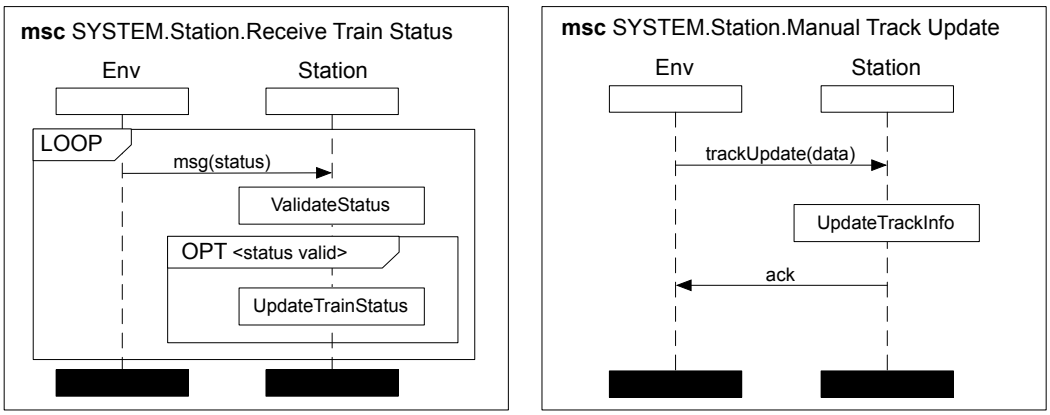

Figure 5.20: BART Station component service interaction specifications

Role Refinement In the Service View, services are defined as interaction patterns between the component and its environment. Subsequently, these services can be refined within the Interaction View. One of the most important refinements targets roles, applying Role Refinement. Figure 5.21 shows an example. The "Station" appearing in the same service in Figure 5.20 now got refined into a "Station Controller" role and an "Env Model" role. The Station Controller performs incoming service message authentication and validation and on success defers to the Env Model component to update and maintain the environmental model keeping track of all trains and other environmental information.

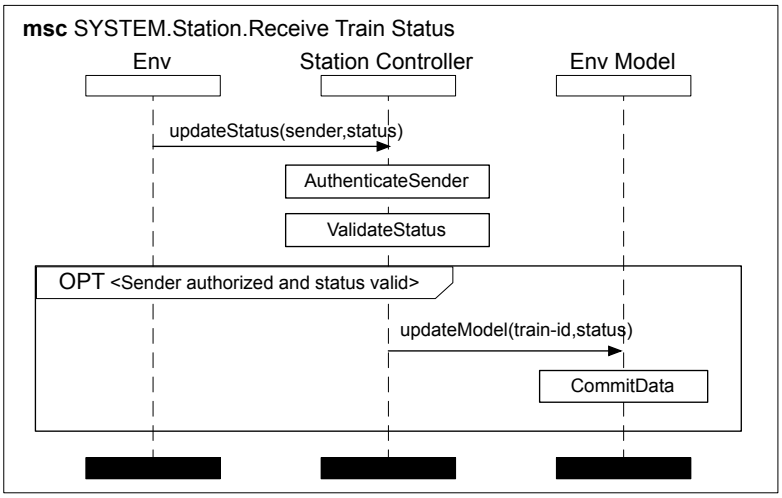

Figure 5.21: BART Station controller "Receive Train Status" refined service interaction specification

Message Refinement in Figure 5.21, we can also see that the message "updateStatus(sender, status)" from the original "Receive Train Status" service specification got refined into a sequence of two messages "updateStatus(sender, status)" and conditionally "updateModel(train-id, status)". This was done along with the Role Refinement. In order to coordinate the interplay of two roles, a new message had to be added causally after the first message, communicating model change information to the Env Model role.

Conditional Refinement Service interaction patterns can also be defined to be valid only in certain operational modes—applying Conditional Refinement. Figure 5.22 shows an example of the "Compute Train Commands" service of the Train component. Per Role Refinement it is defined as an interplay of three roles: "Station Controller", "Env Model" and "Safety Checker". Two of these roles are shared with the specifications shown above. This service interaction pattern only applies to the "Automatic" operational mode of the system, which is inherited by the Station component. Station does not define its own modes, thereby keeping the mode space small.

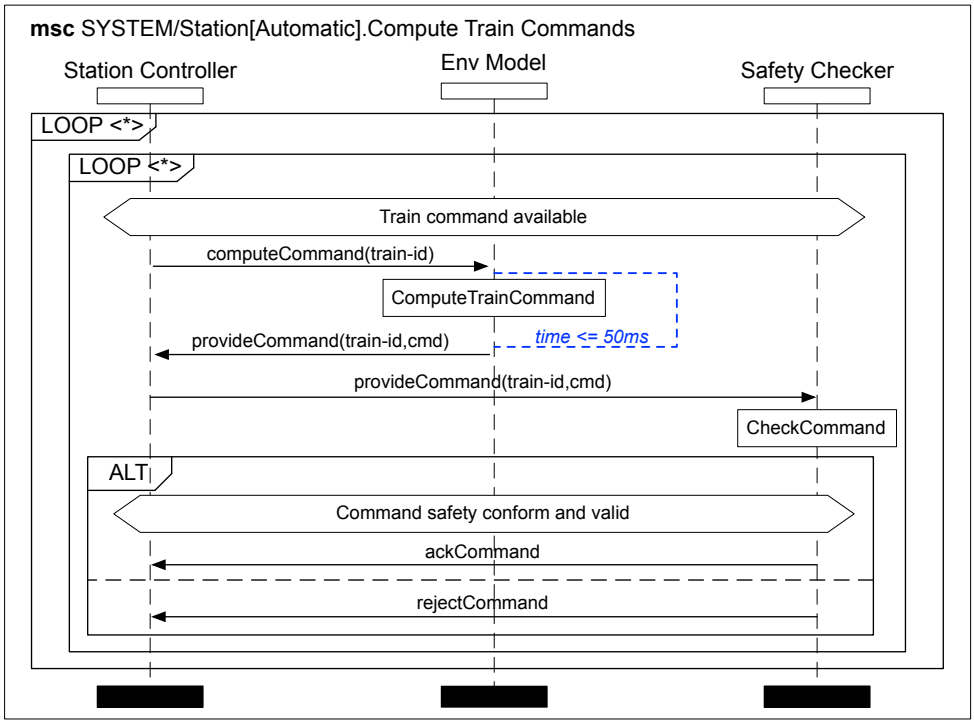

Figure 5.22: BART Station controller "Compute Train Commands" refined service interaction specification

Property Refinement The service specification in Figure 5.21 shows an example of Property Refinement. A local action got added in order to authenticate the sender, a train

controller. Updating the environmental model is conditional on a successful authentication. In addition to its original function of receiving and updating train information, the service now also enforces a security property of only permitting authenticated senders, ignoring any others. This is an example of adding security properties to a distributed system. The communication network is potentially susceptible to spoofing and intrusion by a malicious third party. In order to protect the integrity of the environmental model, security precautions must be taken by the Station controller.

Figure 5.22 shows another example of property refinement. A timing constraint was added that limits the maximum elapsed time between the receipt of a message requesting a train command, and the sending of a response message with the command, to 50 ms or less. Providing such a time bound is essential, in particular for steps such as the "ComputeTrainCommand" local action that have the potential to violate time bounds due to their computational complexity.

Other Refinements Because service interaction specifications and component decomposition are related in our integrated design model, certain refinements of the Structural View also affect the Interaction View. We will introduce refinement notions for the Structural View below.

5.3.2 Refinement of Component Modes

Modes apply to all subcomponents of a component. At any given time, exactly one operational mode applies to the component. The mode automaton for a component can be refined without changing any specifications referencing the modes. The *Mode Refinement* notion applies to the modes and mode transitions of components as maintained in the Modal View of the logical architecture design model. This refinement applies when the designer modifies the number of modes and the mode transitions, without changing the external behavior of the component. This requires that the mode automaton exhibits similar mode transitions after the refactoring. It may have additional modes caused by a splitting of a mode into multiple, and transitions that go with the split.

As discussed above, making a service interaction specification conditional on a mode is a form of Conditional Refinement for interaction specifications. Modes and interactions are related in our integrated design model and one refinement may affect multiple views.

5.3.3 Refinement of Component Structure

Components can be decomposed into subcomponents. This refinement is particularly useful for larger specifications in order to support hierarchical system architecture definition. The following refinement notions apply to the substructure of components as maintained in the Structure View of the logical architecture design model:

- *Interface Refinement:* Modification of the syntactic interface of a component, such as the number of channels and the types of messages. We introduced the formal definitions in Section 3.3.6.
- *Structural Refinement:* Decomposing a component into multiple subcomponents, while maintaining the external interface.
- *Binding Roles to Components:* Mapping roles to subcomponents, conditional to operational modes. Role interaction patterns and subcomponents need to be consistent, for a correct refinement. Consistency and completeness of the specification can be checked algorithmically.

Because services are specified as an interplay of roles that subsequently need to be mapped to subcomponents, there exists a dependency between the Role Refinement of an interaction specification and the decomposition of a component into subcomponents. The mapping of roles to subcomponents provides an indirection; both interaction specifications and component structure can be refined somewhat independently at first, following the most suitable decomposition of component behavior and component structure.

Structural Refinement Component decomposition into subcomponents is essential to develop modular specifications and maintainable system architectures, as explained above. Our formal system model, discussed in Section 3.3.3, provides a strong formalism based on components and a notion of component composition. We will discuss the decomposition of a component into subcomponents in more detail in the following section, in the context of a hierarchical system architecture. We will also show any consequences to related specifications, such as interactions and operational modes, on the subcomponent level.

Binding Roles to Components The diagram in Figure 5.23 shows a *ServDL* mapping of roles to subcomponents for the BART components Train and Station. Some roles, such as the Station's Safety Checker and Timer directly map to subcomponents of similar names. Other subcomponents are composed of multiple roles, such as the Station's Env Model, which plays the roles Env Model and Database. Note that we have not shown interaction specifications for all services in their full extent. The roles appearing in these interaction patterns need to be mapped to subcomponents.

5.3.4 Hierarchical Decomposition of Components

Refining a component structurally by decomposing it into subcomponents is one of the essential system design activities for larger-scale systems. In our development process, the activity "Define component decomposition" supports breaking a component into subcomponents. Our process enables the refinement of a component into subcomponents. The Structure View for a component in our logical architecture design model identifies the

Component Roles SYSTEM.Train
Roles
Train Controller, Timer, Train Control UI, Network Interface
Component Mapping
Car : Train Controller, Timer, Train Control UI, Network Interface

Component Roles SYSTEM.Station
Roles
Station Controller, Controller UI, Env Model, Database,
Safety Checker, Timer
Component Mapping
Station Controller : Station Controller, Controller UI
Env Model : Env Model, Database;
Safety Checker : Safety Checker ;
Timer : Timer

Figure 5.23: Role mapping for the BART Train and Station components

subcomponents and any channels that link them, as well as channels to the environment. The environment, albeit not under the control of system design, can be partitioned into distinct subinterfaces in order to express external responsibilities of subcomponents.

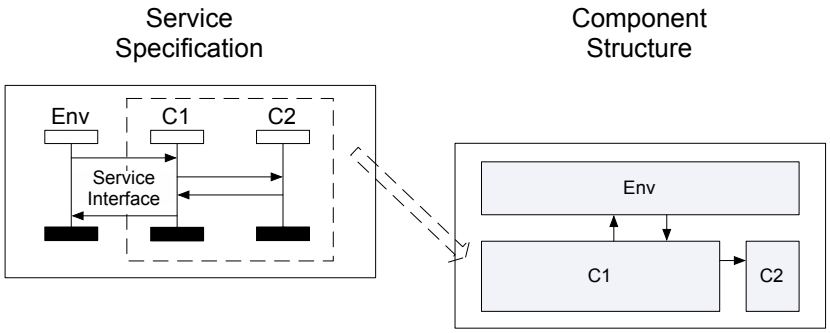

Figure 5.24: Service specification related to component decomposition

Component Decomposition Services constrain component behavior specification and component decomposition in our development process. Services structure a component's functionality towards the environment and the possible interaction of component entities. These entities become candidates for roles in our design model, and as such also for subcomponents. The benefit of service-oriented design is that a component's roles providing services, and the subcomponents are not directly coupled. Many roles can be mapped to a subcomponent, conditional on the operational modes the component is in. One of the consequences of such an integrated service-oriented design model is that component decomposition and component service interactions need to remain consistent, ideally supported by development tools. Figure 5.24 illustrates how Service Specification and Component Structure are interrelated. For matters of drawing simplicity, service specifications use component names as actors, where in our design model role names would be used that can be mapped to component names in a separate step.

Component decomposition can be performed in a modular way if we apply structural refinement. The parts of the design model outside of scope of the refined component remain as is and can be modified in parallel by other designers. Component decomposition can be performed repeatedly. The subcomponents of a component can be decomposed again, as desired. The process activities and work steps can be applied in the scope of the respective subcomponents. Figure 5.25 illustrates hierarchical refinement in our service-oriented process.

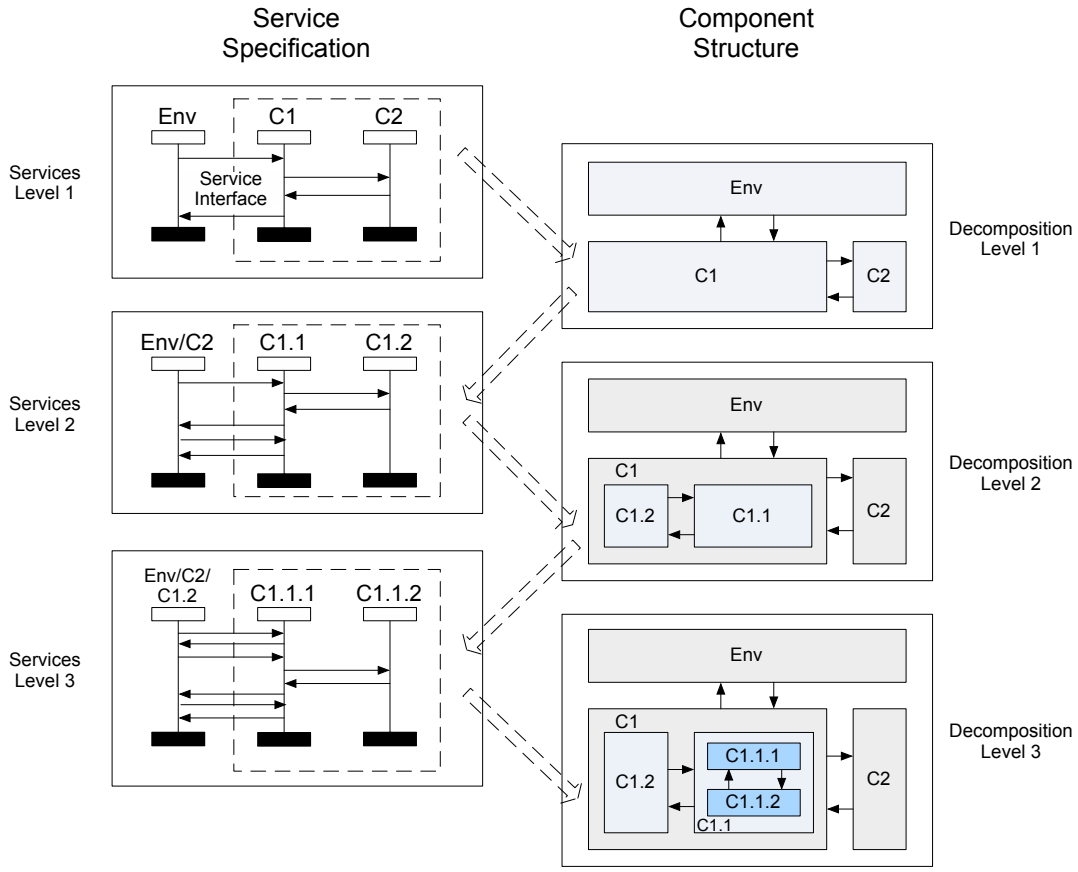

Figure 5.25: Service-oriented system decomposition

Services on the first level drive the decomposition into the first level of subcomponents. Each of these subcomponents provides component level services, which can again be specified as interactions, leading to the second level of component decomposition. This pattern can repeat recursively as needed to break a complex multifunctional component down into a set of maintainable, modular subcomponents.

Figure 5.26 shows the substructure of the BART train car component, consisting of three subcomponents. Not all subcomponents have direct communication relationships.

Figure 5.27 shows the substructure of the BART station controller. This component de-

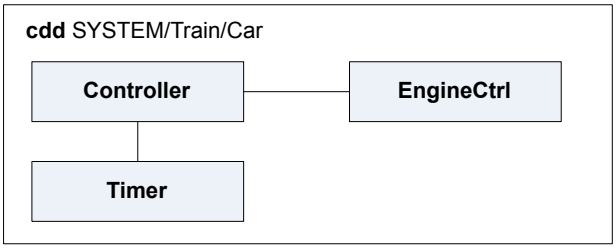

Figure 5.26: BART Train car component decomposition

composition diagram also shows the component's environment as "System Control" and "Network". By looking at the hierarchical system architecture, we know that these two entities are adjacent components within the system; the diagram does not reveal this and remains modular.

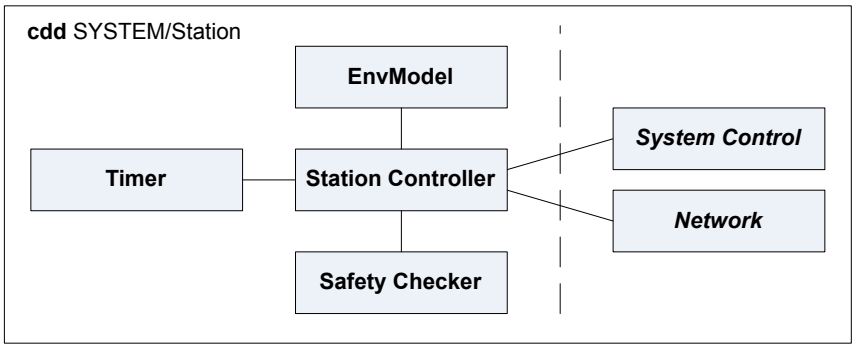

Figure 5.27: BART Station controller component decomposition

5.3.5 Modes for Hierarchically Decomposed Components

Modes apply to all subcomponents of a component, and by refinement to all of their subcomponents. This means that the potential state space of modes is a product of the modes of all parent components. It is not required to make use of this full state space and designers should strive to keep this state space as small as possible when refining components and modes. On the other hand, modes provide the expressiveness to apply service interaction specifications conditionally to when they should be enabled.

Given the operational modes defined by the train car controller component, for instance, as shown in Figure 5.28. This component is on the third level of decomposition under SYSTEM. The modes of the entire system apply, shown in Figure 4.24, in addition to the modes of the system's Train component, shown in Figure 5.14. The Train's Car component does not define its own modes. Figure 5.29 illustrates the state space resulting from the

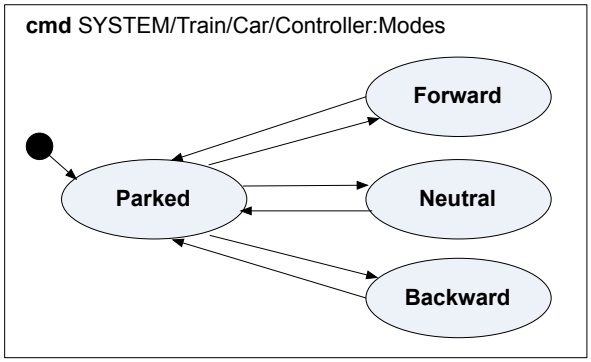

Figure 5.28: BART Train car controller component modes

combination of all the modes for each of the parent components. The total size of the state space is $4 \times 2 \times 4 = 32$.

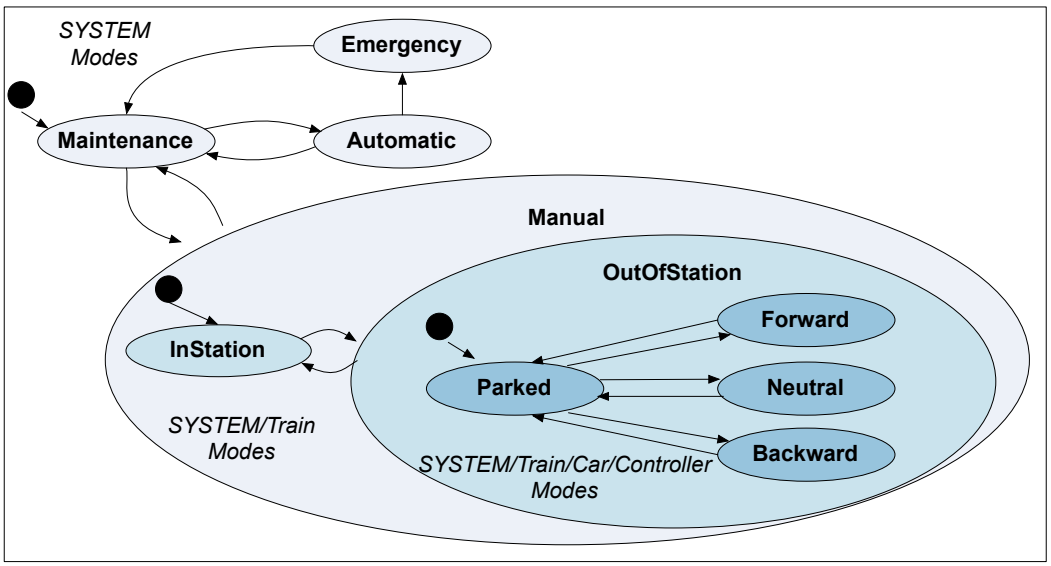

Figure 5.29: BART Train car controller modes state space

It is up to the system designer to leverage the defined operational modes in interaction specifications. There are two basic styles of applying operational modes to subcomponents:

- *Concurrent Modes:* The modes of the parent components do not apply to any of the interaction specifications of a component. The interaction specifications of the component conditionally apply based on the modes defined by the component. The advantage is that the applicable mode state space remains small. Not every component needs to be dependent on a parent's mode. In this case, the component can also be reused outside of the direct context of its parent.

- *Combined Modes:* The modes of the parent components apply to some or all of the interaction specifications of a component. The interaction specifications conditionally apply to a combination of parent component mode and component mode. A disadvantage is that this has the potential to substantially enlarge the applicable mode state space and it also ties the component to the mode state space of its parent(s). If used with caution, however, the designer has full expressiveness to conditionally apply service interaction patterns to specific situations in the system's execution.

5.3.6 Interactions for Hierarchically Decomposed Components

We have shown the interplay of service interaction specifications and component decomposition. Service interactions indirectly constrain component decomposition, which in turn constrains the service interactions on the next lower level of decomposition. There is no "correct" application of Role Refinement and Structural Refinement and other changes to structural and behavioral design specifications. A system designer often has a choice between creating a flatter design with fewer levels of decomposition but higher complexity, or adding layers of decomposition and keeping subcomponents modular and relatively simple. There are several factors influencing these design decisions and often the solution is a carefully balanced compromise between the two goals. In the following, we discuss both variants.

Refining and Extending the Service Model We assume a component with subcomponents. The internal structure of a subcomponent is invisible on the component level. After introducing a new subcomponent, the designer can use it within role mappings, define new service interaction patterns and extend existing ones. New subcomponents share the component data and operational mode definitions.

These model refinements and extensions benefit from the flexibility of the partial service model. New interaction patterns can be added to the set of existing interaction patterns. Changes within existing interaction patterns remain local and thus more simple to perform and verify. A disadvantage of such changes is that any modifications to subcomponent interfaces may cause revisions to the subcomponent specifications. For instance, adding a new service with interaction patterns affects all subcomponents referenced via roles by the interaction pattern. Tool support can help to alleviate the impact of such changes.

Refining Component Structure The inner structure of a subcomponent is invisible on the component level. It is always possible to substructure a subcomponent as long as it maintains its interface. The subcomponent benefits from full isolation from all definitions on the component level. This can be advantage and disadvantage dependent on the context. An advantage is that subcomponents represent modular units that can be independently

developed and verified. A disadvantage is the cost associated with establishing and verifying an interface. It puts a burden on any changes of the component service model going forward.

A Trade-Off Choosing between these two strategies realizes a classic trade-off. When subcomponents can easily be specified in a modular way, need to be broken down due to component complexity, are preexisting or need to be acquired externally, there is no alternative to early component decomposition. Where designs are fluid and are evolving and no decision about component substructure has been made yet, services should be modeled as partial interaction specifications sharing a flat set of roles and subcomponents. Once the set of services and subsequently the set of subcomponents stabilize, the investment in the specification of subcomponent interfaces will be beneficial in order to support large-scale system architectures. Iterative changes will always be possible; their cost will be lower the more contained the changes remain.

5.3.7 Model Refactoring

The term "refactoring" in the context of design models refers to model transformations that change the structure of model elements, without changing any of the properties describing observable behavior.

Refactoring [Fow99] is popular in programming and has gained quite some traction thanks to sophisticated Integrated Development Environments (IDEs), such as IBM Eclipse[1], JetBrains IntelliJ IDEA[2] and JetBrains PyCharm[3] that all apply complex refactoring operations across large code bases. Combined with static code analysis tools, such refactoring operations can even apply to dynamically typed programming languages such as Python and JavaScript. The following transformations realize refactoring operations:

- Extracting a sequence of role interactions as separate MSC
- Combining multiple MSCs into one interaction pattern
- Renaming messages across all interaction specifications
- Splitting a role into multiple roles
- Merging multiple roles
- Splitting an operational mode
- Modifying the data definition applicable to messages or component state

[1] `https://www.eclipse.org/`
[2] `http://www.jetbrains.com/idea/`
[3] `http://www.jetbrains.com/pycharm/`

Some refactoring steps do slightly alter the behavior of the system, e.g. changing the type of message sent as a response to a service request. This may require changes to all callers of the service as part of the refactoring. Some of the changes may be subtle, e.g. when changing the response type from boolean to an integer. In this case, the calling components need to establish additional provisions for dealing with numbers different from the integer values of "True" and "False". As a rule of thumb, the main distinction between a refactoring and a general model change is whether the observable behavior of the system or component changes. In the case of a type change, this is clearly not the case, making it a refactoring operation.

Refactoring for specifications was described for instance in [PR03]. One of the conclusions in that work was that refactorings are a special class of model refinements. The Catalysis approach provides various forms of refinement, abstraction and refactoring patterns for UML based domain and system design models [DW98]. We will discuss Catalysis in more detail below.

Refactoring transformations can be categorized with respect to maintaining these classes of properties:

- *Functional Properties:* The observable behavior of a specification as expressed as a set of functional—or safety—properties does not change. If the inner structure of a specification changes, e.g. the set of roles or subcomponents, it is the externally observable behavior that needs to remain invariant.
- *Timing Properties:* Specified properties constraining the timing of interactions between roles or subcomponents need to remain true. Substituting one message with multiple messages may exceed acceptable time bounds. On order to analyze such properties, a time-based formal system model needs to exist.
- *Liveness Properties:* Such properties express a guarantee of progress in a specification or the absence of deadlocks. A model transformation changing subtle relationships between interacting entities may violate such properties. Model-checking and formal verification techniques can support an automated check of liveness properties given a refactoring step.
- *Quality Properties:* Such properties include security, maintainability, availability and other properties. If they can be expressed formally, they realize one of the above classes of properties or a combination thereof.

5.4 Iterative Service-Oriented Development

Defining a design model for a larger-scale distributed system requires an iterative approach. It is basically impossible to create such a model in one pass. It is similarly unlikely

that during the development time of a project, all requirements, technical constraints and stakeholder expectations can be fully defined early on and kept absolutely stable.

A modular system architecture helps to compartmentalize a system design model and the resulting system implementation, such that changes to the model have limited impact and thereby limited consequences on cost and schedule. Iterating through a process of defined development activities helps to keep the modification steps controlled and somewhat contained, limiting impact and cost of change, and making development more systematic and reproducible. In the following, we describe model development strategies that support such an iterative development process.

5.4.1 Extending the Design Model

Adding elements to a system design model is often the easiest way of advancing it. Extended models often retain many properties and contribute new structure and behavior without modifying existing definitions. A design model defined in a flexible and modular way—for instance by following our service-oriented development process—can be extended with relative ease. Consistent extensions of the model in line with the architect's intent and the underlying architectural principles require substantial thought and care. It is easy to devolve a modular design model into a monolith; it is very difficult to untangle such a design subsequently. A systematic development process and an integrated system design model supporting hierarchical system architectures and partial behavior specifications help to keep design models maintainable, by limiting the amount of change for each development step and by providing model views reducing conceptual complexity.

The following actions are examples of extensions to the design model and can be carried out systematically by design modeling tools:

- *Adding a service:* Service get defined in the Service View as interaction patterns from a service consumer point of view. Subsequently, they get refined in the Interaction View. Services are the primary example of model elements that can be added to an interaction specification without larger impact on the existing model, because they are defined partially. Providing role, data, operational mode and interface specifications of the component are sufficient, new behavior can be added to satisfy new use cases and requirements. The comprehensive—total—component behavior results from algorithmically deriving a joint component behavior state machine from all available service specifications after applying a "closed world assumption".
- *Adding a role:* The addition of a role to a component has no impact on the existing partial service interaction specifications and the existing roles. The new role can be referenced in new interaction patterns, or existing interaction patterns can be refactored or modified to use the new role. This is another benefit of partial service specifications.

- *Adding messages to an interaction pattern:* Existing interaction patterns can be extended with new messages, control structures and state labels. This changes the behavior specification but has no direct impact on other interaction specifications.
- *Adding a subcomponent:* The addition of a new subcomponent leaves existing subcomponents untouched, unless new communication channels need to be added. If subcomponents have not yet been refined and substructured, adding a subcomponent has no further impact on the design model and the new subcomponent can subsequently be used in a revised role mapping.

Our development process provides work steps to perform such model extensions as introduced above. The design model can be checked for consistency after the completion of a work step, published as a new baseline and assessed for consistency across all project artifacts. We will explain this in more detail below.

5.4.2 Model Changes

Other modifications to the system design model have greater consequences and cost. Such changes occur naturally in any larger system design effort and cannot always be avoided.

The following changes design model:

- *Removing a service:* This change has relatively little effect on the system design model because of the partial nature of the interaction specifications. The service needs to be removed together with all detailed interaction specifications in Service and Interaction Views. Removing a service may leave requirements unsatisfied and may also leave parts of component's external interface unfulfilled. The component will remain inconsistent until new or existing services fill the gap.
- *Removing a role:* This change requires the modification of all service interaction specifications in which the role gets referenced. Removing a role is often the last step after modifying interaction patterns, leaving the role unused and ready to be removed.
- *Modifying an interaction pattern:* This includes adding and removing messages, changing message sequences, state labels or control structures, and adding or removing MSC references. Because each service interaction pattern exists as a partial specification, such changes have limited effect on other interaction specifications, but can substantially impact system behavior and the satisfaction of requirements.

As described in the previous section, our development process supports general model changes through defined work steps and artifact consistency assessments. Consequences of general model changes may be larger, requiring more effort in maintaining overall model consistency. Because of the partial nature of our service interaction specifications, changes remain contained, limiting cost and potential for undetected errors.

5.4.3 Systematic Use of Nondeterminism and Underspecification

System design models undergo various maturity stages during system development. At certain points during system development it may become necessary to add new layers of component decomposition to reign in model complexity. The changes related to alleviating such "growing pains" may at first be characterized as a rough cut. At various times during development, design models may exist in an inconsistent or underspecified state. Sometimes this is desirable to facilitate more agile, faster changes to the overall model. Tools can help to support such restructurings and make sure to eventually arrive at an improved consistent model.

An important technique in system design is to keep models intentionally ambiguous, leaving options open for future clarifications. We speak of "nondeterminism" and "underspecification". For instance, when a service interaction specifies the basic flow of messages between a caller role and the roles participating in providing the service, it may not be desirable to fully specify out all the details of the interaction. These details can be added later after the role model has stabilized and no more new services need to be added. Doing so leaves models lean, easier to understand and easier to modify, because refactorings and model changes cannot be avoided.

Once a design model is ready to be matured, it becomes important to be able to detect cases of ambiguity. Our formal system model, for instance, allows us to analyze interaction specifications to detect nondeterministic behavior. It is also possible to verify whether a component interface gets completely satisfied by the existing set of service specifications. Once underspecification gets detected, the designer can subsequently work on adding or refining model elements to make the behavior model causal and thereby ready to be implemented.

5.4.4 Managing Design Model Change

We have introduced several categories of design model change from refinement and refactoring to extension and other modifications. Some of these changes have guaranteed—limited—consequences, such as refinements. Other changes may leave the design model inconsistent or may cause violation of certain system properties. Our development process provides activities and work steps to limit the impact of individual changes to the larger model, by keeping model changes contained, for instance to a single component or service, and by limiting the permissible change within one work step.

Our development process supports analytic and constructive quality assurance activities after or in between model changes. The following mechanisms help to keep the system design model consistent:

- *Artifact tracing:* This constructive quality assurance mechanism ensures that related artifacts and model elements remain linked, for instance enabling quick impact

analysis of proposed changes. Tracing needs to be added after the addition of new model elements. Modifications may require changes to tracing. The integrated logical architecture design model provides certain tracing links through its metamodel, for instance the links between a component and its subcomponents, or between an interface and a service implementing part of the interface. Other traces, such as components to requirements need to be manually added and maintained.

- *Internal model consistency:* Our integrated logical architecture design model supports the detection of unwanted inconsistencies, given its precisely defined metamodel. Model elements defined in one model view, for instance the Data View, and referenced in a different model view, for instance the Interaction View, can be automatically analyzed. The designer has the ability to leave the model temporarily inconsistent, for instance by adding unspecified message types and arguments to an interaction pattern, but will eventually be required to add the necessary data element specifications to the Data View in order to make the model consistent again.
- *Consistency checking:* Our development process artifact model defines consistency rules across process artifacts that can be evaluated as required, for instance after a set of model changes. Artifact tracing combined with model element tracing is often a prerequisite to determining the applicability of consistency rules, especially given our modular, hierarchical system model. Checks can be limited to specific artifacts or system elements.

Our process requires to quality control artifacts before they can be baselined and publicized, as for instance required in order to provide a component specification to a subcontractor for implementation.

5.5 A Service-Oriented Extension of the V-Modell XT

In this section, we show the integration of our service-oriented development process with the V-Modell XT [Bun12]. The goal is to utilize the proven project management, quality assurance and system engineering frameworks of the V-Modell, combined with the precise methodology that service-oriented development provides. We apply the V-Modell extension mechanisms to embed our core process seamlessly within the process framework. We published an earlier version of this integration in [MK07]. An integration with a comparable process framework such as the Unified Process [JBR99] can be done accordingly, if desired.

5.5.1 Modular V-Modell XT Extension Mechanism

In this section, we describe the V-Modell extension mechanism that we will subsequently utilize. The V-Modell's core process elements include work products, activities and roles.

Work products and activities are directly related; both are associated with a discipline, such as "System Design". Work products substructure into subjects, activities into work steps. Roles are related to work products. Work products have dependencies on other work products. Figure 5.30 illustrates the process elements and their relations. Section 2.5 provides a more comprehensive description of the V-Modell XT.

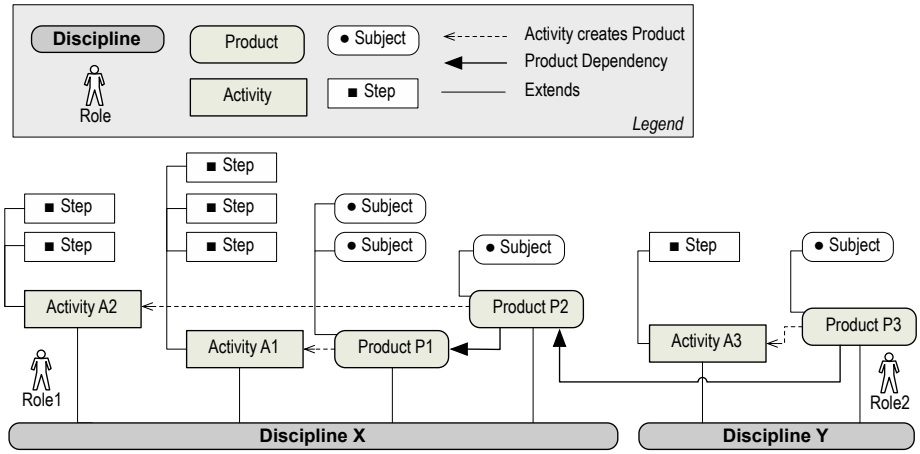

Figure 5.30: V-Modell XT process elements and relations

The V-Modell's main structuring and extension mechanism is the process module. Each process module contains all process elements required to support a specialized process, such as the specification of requirements and hardware development. Process modules define work products, activities and roles associated with existing disciplines. Process modules can also contribute extensions to work products and activities defined in other process modules in form of additional subjects and work steps. Further contributions include product dependencies, method references and tool references. Adding a process module thereby may grow the extent of a work product. A similar mechanism applies in reverse when tailoring the V-Modell XT for a new project. First, you add the needed coarse building blocks based on a "cafeteria style" choice of process modules and selection guidelines, and then you remove superfluous detail by pruning work products and activities. Both steps are considerably easier than authoring or extending a process afresh and may in many cases not even be necessary.

Figure 5.31 abstractly illustrates the extension of V-Modell process elements defined in two existing process modules—the blue "Base" and the green "General" module—with elements of a third red "Extend" extension module. The process elements are color coded by process module and stacked vertically according to their extension hierarchy in order to clearer illustrate the dependencies.

The successive extensions of the V-Modell core process modules result in a process module dependency graph, as partially depicted in Figure 2.14. More specific process modules extend required basic process modules. Tailoring occurs after the extension. Some process

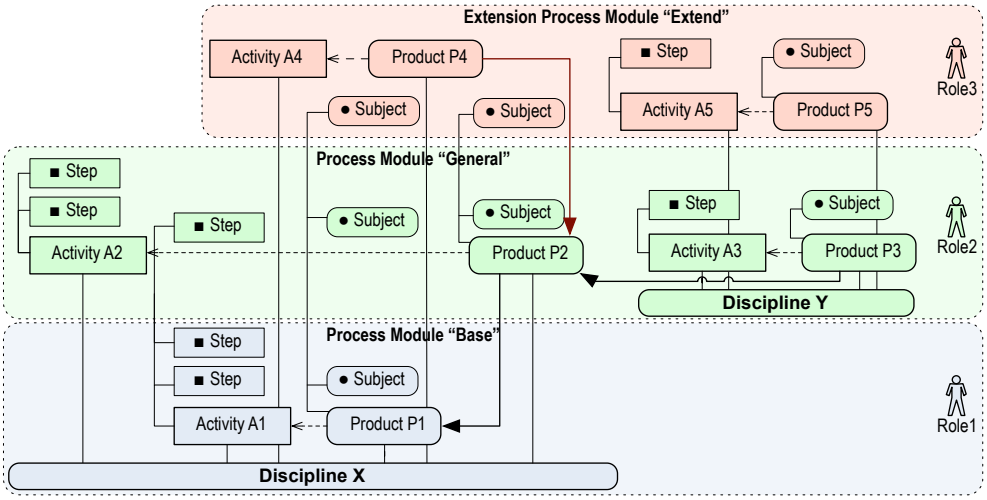

Figure 5.31: Extending of V-Modell XT process elements through process modules

elements coming with the extensions can be tailored away subsequently, if not applicable to a project. For instance, extensions to hardware development activities can be removed in case the project does not develop hardware. Note that process extension and tailoring both happen before the process is instantiated for a new project. Figure 5.32 shows the same process as Figure 5.31 without the context of the process modules. The five artifacts are highlighted indicating their coherency. Color coding indicates process module origin. A process resulting from extension is a coherent, consistent whole. Any individuals working with an instantiated process need not be concerned about how the process was extended and tailored.

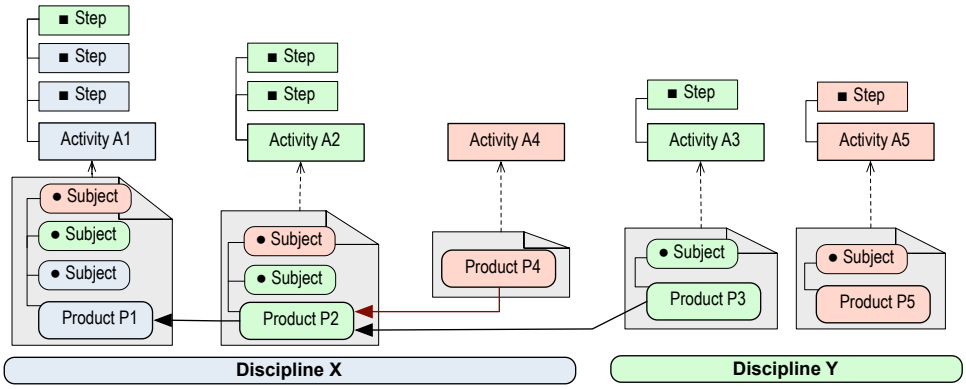

Figure 5.32: Resulting V-Modell XT process after extension

The V-Modell XT in its current version defines 22 process modules, with a mandatory core of four process modules. The addition of further process modules follows exactly the same mechanism and once instantiated cannot be distinguished from the rest of the

process framework. Third-party extension are not part of the standard "package" of the V-Modell XT—third parties are responsible for the correctness and consistency of their additions.

The rigorous V-Modell XT metamodel [Bun12] defines all process element types and their relations. It is the basis for consistent extension and tailoring mechanisms, and for automating these tasks through tool support. Tools exist that make use of these definitions, such as the "V-Modell XT Project Assistant", based on the concepts introduced in [MRS06, GMP+03].

5.5.2 Process Module Service-Oriented Development

The V-Modell XT provides comprehensive coverage of the system engineering activities, including requirements specification, architecture design, implementation, integration and verification of software systems and their components. The level of detailed method guidance is rather low, given the wide range of projects and development styles that the V-Modell applies to.

In this section, we show how to extend the V-Modell with specific guidance for service-oriented development, realized in work product, product dependency and activity definitions, and their detailed descriptions.

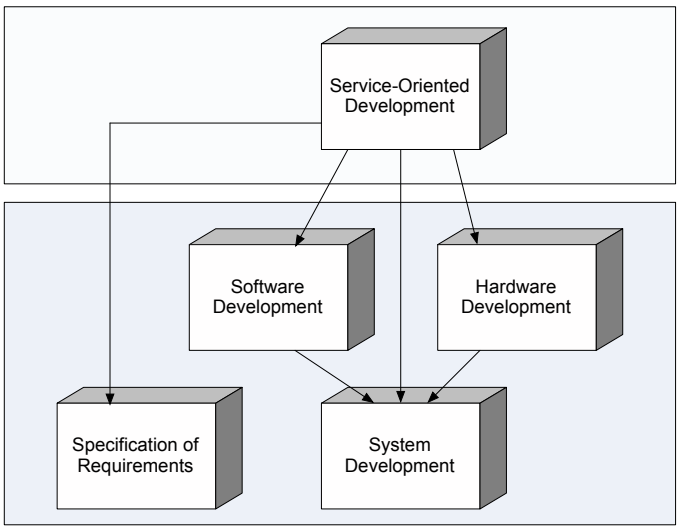

Figure 5.33: Service-oriented V-Modell XT extension

We define a new process module "Service-Oriented Development" as depicted in Figure 5.33. This process module is based on the V-Modell modules "Specification of Requirements", "System Development", "Software Development" and "Hardware Development". Because the V-Modell covers basic system engineering activities sufficiently, we only need

to extend existing work products and activities instead of adding new ones. We make use of the fine-grained extension mechanism explained above by adding new subjects to existing work products and new work steps to existing activities. Additionally, we add product dependencies to keep our additions consistent with the rest.

Artifact Model The hierarchical decomposition of the system into system elements is mirrored by V-Modell products and activities. One instance of a work product exists for every major system element, accompanied by the execution of a corresponding activity. This modular, hierarchical pattern applies equally to service-oriented development, as explained in depth above. Each element of the system architecture—with existing specification and architecture documents—additionally defines a service model with tracing links to the existing artifacts. Thus, our service-oriented approach inherits the scalability benefits of the V-Modell XT.

Due to project organization and the distinction of different types of system elements, the V-Modell XT defines different work products for similar classes of artifacts. We will reference them by their "class" only:

- *Requirements documents:* The V-Modell work products "Requirements Specification" and "Overall System Specification" have a similar structure but differ with respect to the responsibilities in the acquirer–supplier project relation. Both work products include requirements for the overall system as well as additional information to understand system context and scope of the development effort.
- *Specification documents:* The work products "System Specification", "Software Specification", "Hardware Specification", "External Unit Specification", "External Software Module Specification", and "External Hardware Module Specification" realize specification documents that focus on interfaces and their properties in order to provide a black-box view on a system element. All these work products have similar structure with minor differences caused by the nature of the system element.
- *Architecture documents:* The work products "System Architecture", "Software Architecture" and "Hardware Architecture" realize architecture documents that show design alternatives and design details, including decomposition, internal interfaces, data types and behavior. All these work products have similar structure with minor differences caused by the nature of the system element.
- *Construction plans:* The work products "System Implementation, Integration and Evaluation Concept", "Software Implementation, Integration and Evaluation Concept", and "Hardware Implementation, Integration and Evaluation Concept" are plans to describe procedures, schedule, structure and oversight of system construction.

Table 5.1 shows a mapping of the elements of our service-oriented artifact model to work products and subjects of the V-Modell XT. Subjects are indicated in italics. Some of

our process artifacts map to existing V-Modell work products and subjects, meaning they will be delivered as part of the mapped V-Modell process elements. V-Modell elements marked with an asterisk (*) in the table are extensions provided by the new process module "Service-Oriented Development". These are contents not presently defined by the V-Modell. As mentioned, we do not define new work products and rather extend existing work products with new subjects.

Product Dependencies The consistency across work products is ensured by product dependencies. We add the following dependencies:

- *Consistent Service Refinement:* Dependency between products *Overall System Specification* and *Architecture*: the identified interfaces and services for one system element realize the specified requirements, use cases and nonfunctional requirements.
- *Consistent Service Model:* Dependency between products *Specification* and *Architecture*: All services defined in the specification must be consistent with interfaces; all services have interaction pattern specifications; interaction patterns must only use defined roles and operational modes.
- *Consistent Data Model:* Dependency between products *Specification* and *Architecture*: Interface, service and interaction specifications must reference elements specified in the data view.
- *Consistent Role Model and Mapping:* Dependency between products *Specification* and *Architecture*: Role definitions can only refer to defined subcomponents and modes; the subelement structure as given by the decomposition must be a refinement of the role model; all roles and services must be mapped to subelements

Activities At the top level, the V-Modell defines a number of disciplines, which correlate roughly to our high-level activities. Table 5.2 provides a mapping of the four high-level activities of our service-oriented process to V-Modell disciplines. Because the mapping is not one to one, the table specifies while activities and work steps are targeted. A mirrored mapping exists for artifacts instead of activities.

The activities of the V-Modell XT mirror work products. Each work product has exactly one associated activity. Because we don't add new work products, there are also no new activities defined by our process module. Instead, we provide work steps that describe how to create the new subjects of the work products, and how to keep them consistent. The work step definitions follow our explanations of Section 5.1.

Table 5.3 shows a partial mapping of activities of our service-oriented process to activities and work steps of the V-Modell XT. Work steps are indicated in italics. V-Modell elements marked with an asterisk (*) in the table are extensions provided by the new process module "Service-Oriented Development".

Table 5.1: Artifact model to V-Modell work product mapping

Artifact/View	**V-Modell Element**	**Description**
Requirements Artifact	Requirements Document	Context, scoping, functional and nonfunctional requirements.
System Specification	System Specification: *Interface Specification, Nonfunctional Req.*	Precise specification of functional and quality properties of the system.
Designed System	Architecture Documents	Full logical system design
Interface View	Specification Document: *Interface Specification*	Defines externally accessible interfaces.
Service View	Specification Document: *Interface Realization, Service Access Points**	Refined interfaces for service access.
Structure View	Architecture Document: *Decomposition, Interface Overview*	Identification of subelements and their interfaces
Modal View	Architecture Document: *Overall characteristics*	Operational modes and transitions.
Data View	Architecture Document: *Data Catalog**	Identification and structure of messages, data types etc.
Role View	Architecture Document: *Role Definitions**, *Role Mapping**	Definition of roles as component abstractions. Mapping to components in modes.
Interaction View	Architecture Document: *Service Specifications**	Specification of interaction patterns for services
Component Behavior View	Architecture Document: *Component State Model**	Full component behavior specification.
Deployed System Design	System Architecture, Construction Plans	Full logical and technical system design and construction plan.
Component Configuration	Construction Plan *Realization Procedures, Integration Procedures*	Assemblies of components as result of integration.
Deployment Mapping	Construction Plan *Installation Procedures and Target Environment*	Target environment characteristics and deployment plan.
Implemented System	System	The installable system
Test Artifact	Evaluation Procedure	
Build Environment	Enabling System	One particular enabling system

Table 5.2: High-level activity to V-Modell discipline mapping

High-Level Activity	V-Modell Discipline and Elements
Requirements Engineering	Requirements and Analyses, System Specifications: *Work steps related to evaluating and revising system requirements, refining requirements for system elements. Defining system scope, life cycle and acceptance criteria.*
Logical Architecture Design	System Specifications: *Work steps related to preparing the overall system architecture. Tracing of requirements.* System Design: *Work steps related to preparing and defining the system architecture for system elements and for preparing logical designs.*
Technical Architecture Design	System Design: *Work steps related to preparing the integration concept and developing technical designs.*
Implementation and Integration	System Elements

Roles The integration of our service-oriented approach enhances the responsibilities of five roles that already exist in the V-Modell XT. Their role profile descriptions are sufficiently abstract and fit our purposes. Thus, only slight extensions need to be performed:

- For the roles *Requirements Engineer (Acquirer)* and *Requirements Engineer (Supplier)*, we add domain-modeling and service-oriented design as required capability.
- For the roles *System Architect*, *Software Architect* and *Hardware Architect*, we require knowledge of service-oriented design and of service-oriented architectures and infrastructures.

5.6 Summary

This chapter has provided a comprehensive service-oriented development process with particular emphasis on the logical architecture design activities. The process defines activities and work steps and relates them to the artifacts defined in the previous chapter. We provided structure to the process by associating activities with the abstraction levels applicable to service-oriented model-based system design. We provided activity flow diagrams showing logical input-output relationships of activities as part of the process. Where illustrative we provided references to our running example.

Table 5.3: Activity to V-Modell activity mapping

Activity	**V-Modell Element**
Define component interfaces: *Define external interface*	Preparing Specification: *Identifying Interfaces and Nonfunctional Requirements,* *Refining Interfaces and Nonfunctional Requirements*
Define component interfaces: *Trace to requirements*	Preparing Specification: *Allocating Interfaces and Nonfunctional Requirements*
Define component interfaces: *Define data entities and flow*	Preparing Architecture: *Defining Data Elements**
Define component interfaces: *Define component services*	Preparing Architecture: *Defining Service Interfaces**
Define component service model *Identify roles*	Preparing Architecture: *Defining Roles and Role Structure**
Define component service model *Define service interactions*	Preparing Architecture: *Defining Service Interaction Patterns**
Define component service model *Define operational modes*	Preparing Architecture: *Defining operational modes**
Define component service model: *Define component decomposition*	Preparing Architecture: *Preparing Architectural Views*
Define total component behavior *Map roles to components and modes*	Preparing Architecture: *Mapping Roles to System Elements**
Define total component behavior *Generate total component behavior from interaction patterns*	Preparing Architecture: *Defining total component behavior**

A particular focus of this chapter has been the applicability of our process in larger-scale project settings with sizable teams, a convoluted domain and requirements space, a high degree of parallel work due to the size of the effort, and multiple planned incremental and iterative revisions of project results. Our process supports iterative system development. Core activities expose both a natural flow and can be iteratively applied. We showed various forms of design model modifications, including model refinement and refactoring operations, and related these to the definitions made when we introduced our formal system model. The importance of a modular, hierarchical system architecture became very clear in the context of our development process. It directly supports flexible iterative artifact revisions with limited effort to maintain artifact consistency and with contained risk of undetected consequences of model changes.

We showed an integration of our development process with the existing V-Modell XT system engineering process framework. This makes our process readily accessible to projects looking for comprehensive, tailorable process support. The V-Modell XT provides existing tool support that can be leveraged. By design, the V-Modell supports extensions with specialized process modules that seamlessly integrate into the existing activity and artifact structure. We laid out the integration of our process as part of a "Service-Oriented Development" process module.

6 Evaluation and Discussion

In this chapter, we evaluate our service-oriented development process. We describe an effective tool architecture showing how our process can be automated through tool support. We describe existing tools and a proof of concept implementation for an integrated tool landscape. This demonstrates the practical applicability of our solution and its scalability. We address the problem statements made in the motivation. We evaluate our approach against the key properties of distributed reactive system and service-oriented development approaches. We also compare our approach against existing development approaches. Finally, we present findings from applying service-oriented development in practice and list known shortcomings of our solution.

Contents

6.1 Evaluation Strategy

Evaluating the effectiveness of a development process is challenging. It is not possible to exactly compare different development processes for the same set of project circumstances, such as requirements, project constraints, available resources, available personnel, and existing technologies. The same development cannot be repeated under different circumstances. Comparing individual characteristics in isolation does not address the sophistication of most approaches.

In academic practice, the preferred evaluation method for a proposed new methodology is to compare its application with an existing approach using an identical system. First, analyze the application of an existing approach to develop a well characterized system, determining benefits and shortcomings. Subsequently, apply the new approach to the same system, and show that the new method has enhanced benefits, while not significantly increasing the shortcomings, for the applicable system class. For our service-oriented development process, we do not have a complete system development project that we can compare against a similar development with an existing approach. Nonetheless, the following available strategies exist to evaluate the effectiveness of a development approach consisting of theory, methods, tools and processes:

- *Qualitative analysis:* Analyzing the degree to which the process and applied techniques satisfy previously identified key properties. Using a realistic case study is critical to demonstrate that process properties are satisfied.
- *Qualitative comparison:* Comparison with alternative development processes and applied techniques based on previously defined characteristics and requirements. A common case study helps to evaluate different approaches more consistently.
- *Quantitative comparison:* Comparing consistently collected metrics across approaches. Metrics need to be defined objectively and the system under development needs to be similar.
- *Ability to automate:* Demonstrating the possibility of automation and showing the existence of tool support are strong indicators for the practical applicability of a solution.
- *Comparability:* Providing a robust mapping to another approach and arguing that the approach under evaluation is at least as capable as the other approach.
- *Empirical study:* Comparing characteristics of approaches using published empirical evidence as basis of comparison. Requires evidence for isolated characteristics that can be attributed to the approach under evaluation.
- *Proof:* Formal reasoning the correctness and completeness of a theory, based on a comprehensive theoretical model.

For this thesis, described in the following sections, we demonstrate the ability to automate our approach, qualitatively analyze our solution against previously established key properties, and qualitatively compare with selected existing approaches using a realistic case study.

6.2 Automating the Service-Oriented Development Process

Effective automation through tool support helps to demonstrate the applicability of an approach. In this section, we name existing tools supporting our service-oriented development approach. We introduce an effective architecture for comprehensive process support.

Automation and tool support are essential for users to utilize the powerful abstractions and notations of more sophisticated approaches such as ours. In particular, working with an integrated system design model providing various cross-linked model views requires assistance. Such tools can support model authoring, find inconsistencies in specifications, map design models to target architectures, generate prototypical implementations and verify implementations against the specification.

6.2.1 Requirements for Process Automation and Tool Support

The following high-level requirements characterize an effective automation of our service-oriented development process. Tools supporting our approach can be measured against these requirements.

- *Cover the entire process:* Tool support shall span all development activities from requirements engineering to system deployment. The emphasis shall be on service-oriented system specification and architecture design.
- *Represent process artifacts:* Tool support shall support all defined the artifacts of the process artifact model. Artifacts shall be viewable and changeable through the tool.
- *Maintain artifact tracing:* Tool support shall maintain and manage dependencies between artifacts and where applicable, between entities specified within artifacts, such as system elements within a design model. Tool support shall keep track of system engineering basic traceability between artifacts, such as from requirements to design and implementation.
- *Assert consistency:* Tool support shall enable the definition and evaluation of consistency checks that cover the represented artifacts and their dependencies. Tool support shall provide basic artifact consistency checks and ensure artifacts are consistent when a baseline gets published.

- *Compute metrics:* Tool support shall enable the definition and computation of metrics based on the managed artifacts. Metrics could include indicators about the level of completion of system development, the quality of the tracing, or the complexity of system designs.
- *Transform artifacts:* Tool support shall enable the definition and execution of transformations targeting the managed artifacts, for instance to execute refinement and refactoring operations.
- *Support extension:* Tool support shall enable the configuration and execution of plug-ins accessing the represented artifacts, for instance for purposes of verification and code generation. The tool shall realize an extensible platform.
- *Ease of use:* Tool support shall enable users to execute common tasks effectively and intuitively without requiring extensive training or study of the documentation. Documentation shall be provided as necessary, providing telling examples. Automation for repetitive tasks shall be provided.

In the next section, we introduce an architecture for tool support with the potential to satisfy all of the above listed requirements. An actual implementation can then be measured against these requirements.

6.2.2 A Comprehensive Process Automation Strategy

Tool Architecture Our goal is to automate all activities of our development process supported by one tool. The tooling should provide a platform for the integration and application of more specialized methods and tools. The platform should be able to flexibly represent all process artifacts and support the process activities.

Figure 6.1 shows the architecture of a tool landscape with the SODA (Service-Oriented Design and Architecture) tool as process platform, cf. [EHK+07]. SODA embeds editors for system architecture and design model artifacts, such as service interaction specifications using MSCs, and system architecture definition tools. SODA manages the model artifacts and provides capabilities to execute plug-ins that perform model consistency checking and transformation, for instance for refinement and refactoring purposes. In addition, SODA triggers specialized tools that perform model-checking of properties, prototype code generation, design model export and architecture exploration.

The main use scenario for SODA is illustrated in Figure 6.2. The tool supports the entire development process from requirements engineering—for instance use case definition—to system architecture definition, service specification, and component implementation generation.

SODA provides capabilities supporting project management during the development process, such as task and resource tracking, project planning and document generation.

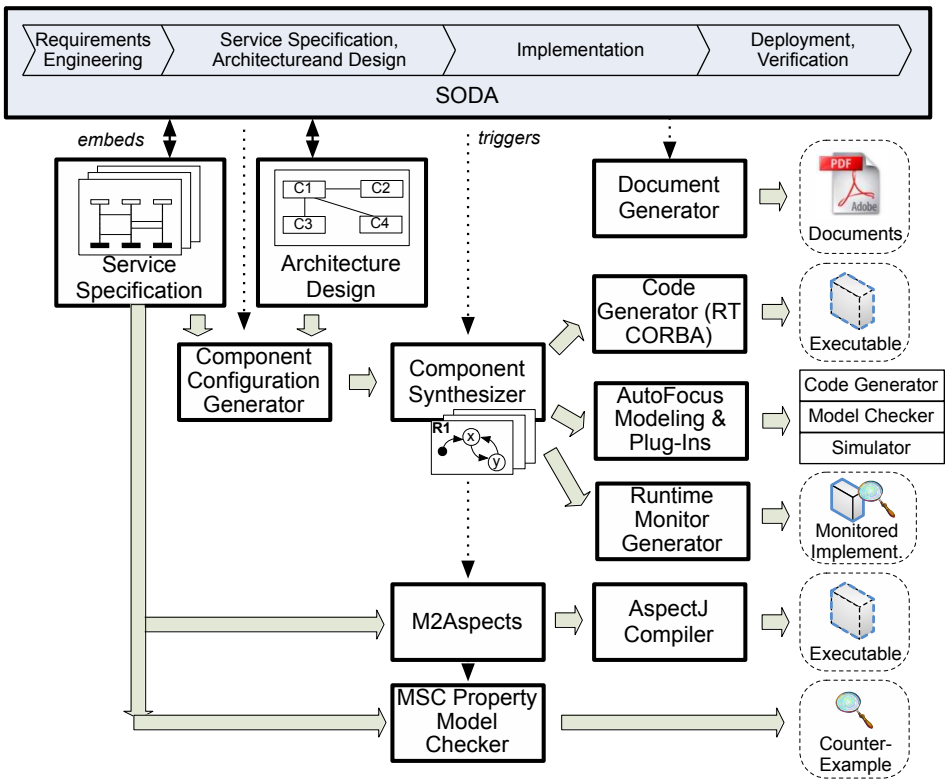

Figure 6.1: Automation and tool support architecture

Through plug-in document exporters, SODA can generate requirement specifications, architecture documents, management reports and any other derived artifacts based on the underlying artifact repository. SODA maintains a structured artifact repository and manages consistent cross-references between the elements within the artifact repository as well as to external artifacts, such as document references within a local network.

The SODA Prototype We have implemented a proof of concept of SODA to demonstrate the feasibility of the architecture. The implemented design satisfies requirements for flexibility and extensibility. At the core of the SODA implementation is a highly flexible generic structured object model framework, 4ever[1] [MRS06]. 4ever uses an XML-based persistence layer. An XML schema defines the structure of the artifact repository including artifact dependencies and basic consistency rules. 4ever provide an abstract representation of the object repository and maintains bidirectional cross-references between objects. It also automatically maintains repository consistency. The artifact repository can be configuration controlled using tools such as Subversion[2] and GIT[3].

[1] `http://fourever.sourceforge.net/`
[2] `http://subversion.apache.org/`
[3] `http://git-scm.com/`

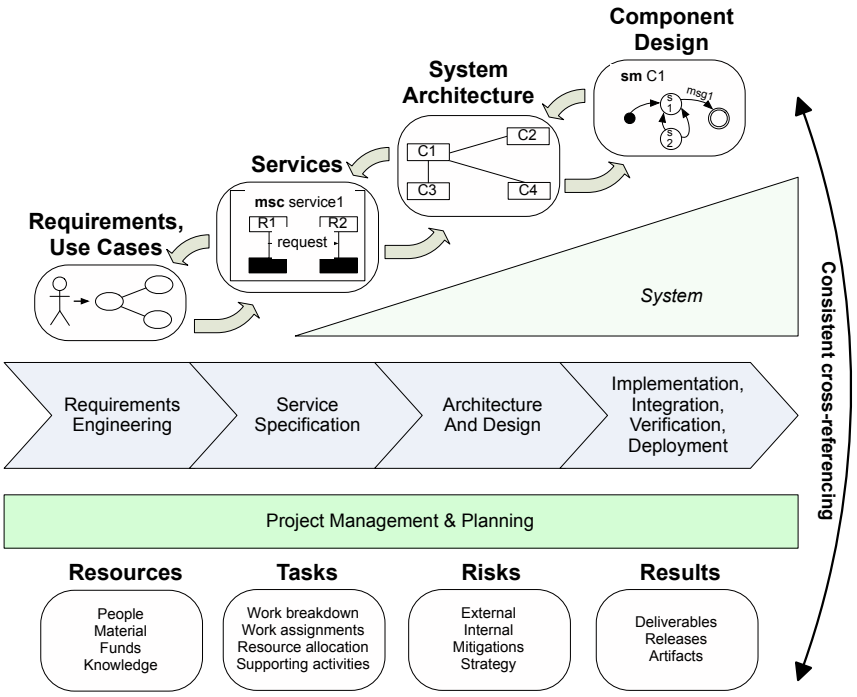

Figure 6.2: SODA tool use scenario

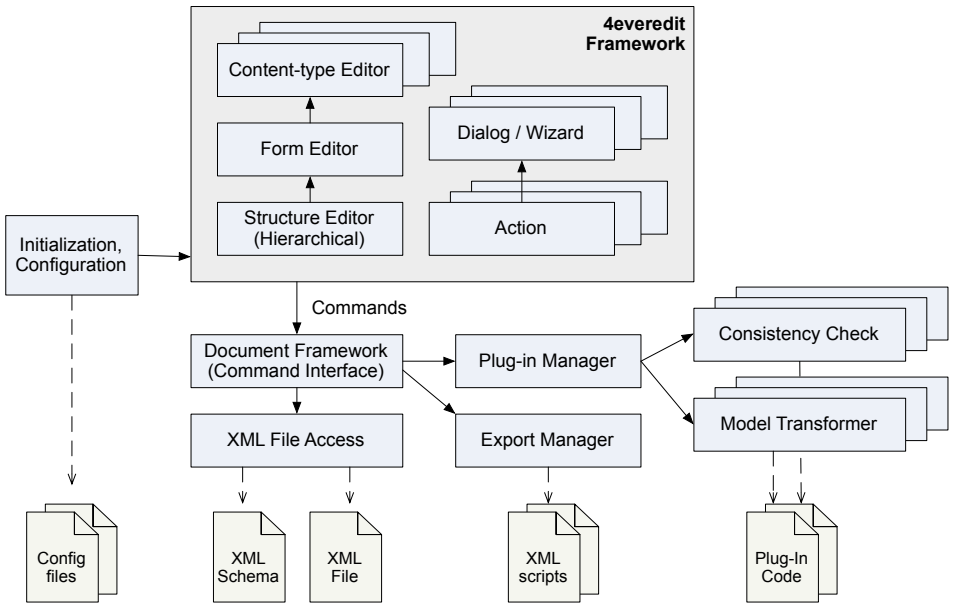

Figure 6.3: SODA tool implementation architecture

4everedit is an open source graphical editor component based on 4ever. It is highly generic and can represent arbitrary content of any structure. 4everedit is also flexibly extensible with output generators, consistency checkers and artifact editor elements. The entire V-

Modell XT process documentation—described in Section 2.5—was authored and generated using a customized version of 4everedit. This illustrates its practical applicability and scalability.

Figure 6.3 shows the core components of the 4ever framework and 4everedit as well as extensions provided by SODA. SODA contributes a service-oriented process metamodel and artifact model. Additionally, it enables the definition and execution of model consistency checkers and model transformers. It defines an abstract object model that represents the core artifacts and dependencies of our service-oriented artifact model.

Figure 6.4 depicts a screenshot of the proof of concept implementation, showing the specification of the "Receive Train Status" service of the BART AATC case study. The service specification includes references to used roles and states (not shown), a service name and description and a graphical specification of the service as MSC.

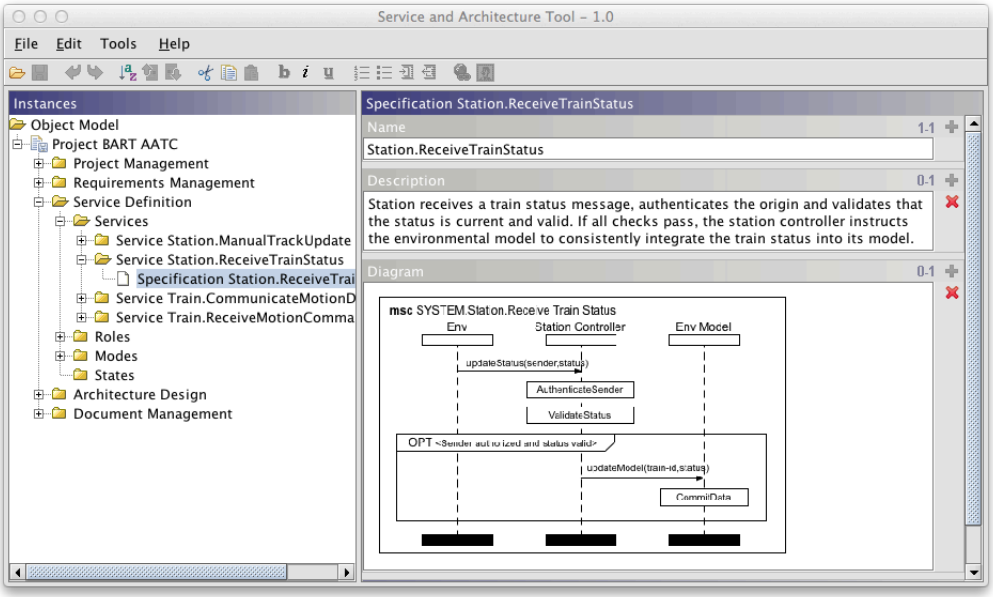

Figure 6.4: Defining services in the SODA tool

Figure 6.5 depicts a screenshot of role specifications for the same project repository. The tool shows references to other model entities, such as states. The screenshot also shows the menu options to execute consistency checks across the model.

Figure 6.6 depicts a screenshot with the resulting output from running role level consistency checks on the model repository. The tool detects inconsistencies in the model, such as states that have not yet been referenced by other model elements. This could indicate a problem in the model, for instance resulting from a prior manual edit step, or just work in progress with issues to be fixed before the next consistent snapshot.

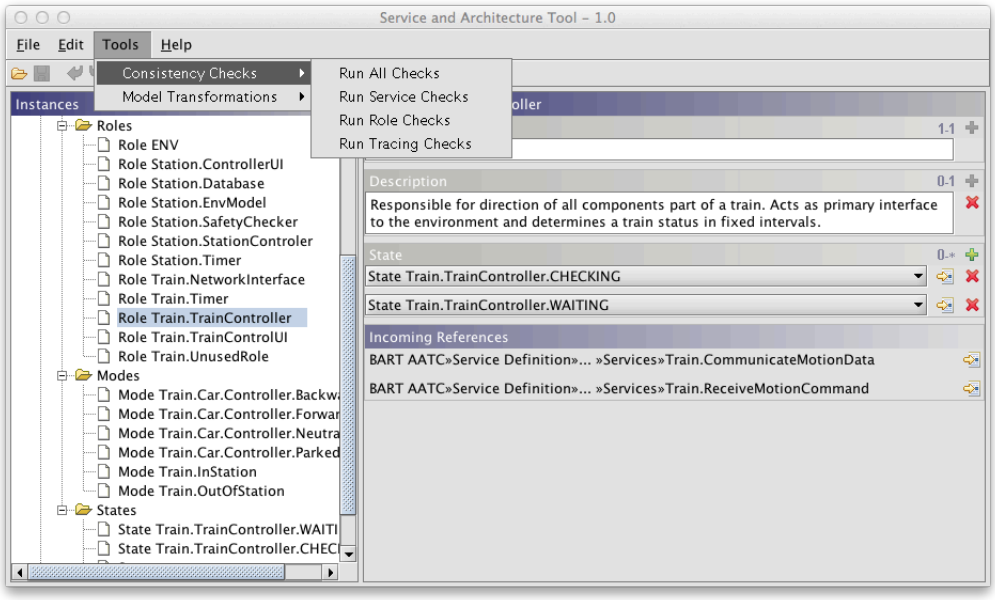

Figure 6.5: Defining and cross-referencing roles in the SODA tool

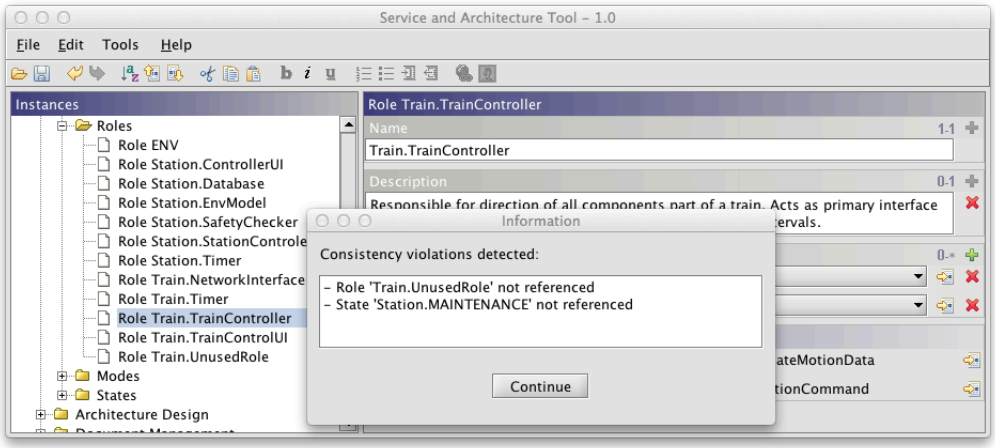

Figure 6.6: Executing consistency checks in the SODA tool

Integrated Tools A number of tools exist and can be integrated into the SODA process support platform. *M2Code* is a tool to comfortably edit MSC-based interaction pattern specifications. It can represent a consistent hierarchy of Basic MSC and High-level MSC specifications. Powerful operators as introduced in Section 3.3.2 support service designers in structuring the service repository.

A core tool in our tool chain is the *component synthesizer*. It takes MSC-based service specifications as input and generates component state machines for all components referenced in the interaction patterns. This tool is based on the state machine synthesis algorithm

described in Section 3.3.5. It's output can be used by other tools.

Tools taking generated component state machines as import include the RT CORBA code generator [AKMP05]. It generates executable distributed systems based on a Real-Time CORBA middleware infrastructure. Such prototypes can be used for architecture exploration, component compliance testing and for checking basic quality properties of the distributed system, such as interaction deadlines.

Another converter takes the state machine specifications and makes them available to the CASE tool AutoFocus[4] [HS97]. AutoFocus provides a practice proven platform for component-based embedded real-time system design with a semantics foundation based on the FOCUS theory [BS01]. AutoFocus enables the refinement of the state machine specifications as well as provides connections to a multitude of plug-ins, including Java code generators, model checkers, event trace generators and system simulators. It also provides access to theorem-proving tools.

We also prototyped a powerful runtime monitor generator [KMM08]. It takes component state machine specifications and generates runtime monitors for existing applications. Applications include stand-alone Java applications as well as Enterprise Service Bus (ESB) [KMMP06] deployed applications plug-ins that communicate using message passing middleware. We used aspect-oriented techniques to augment existing executable code with runtime monitors. These monitors observe the interactions of the distributed components and raise errors in case of protocol violations.

We worked on an AspectJ[5] code generator to generate executable prototypes of distributed service-based system specifications. We used these prototypes for efficient architecture exploration [KMM06] and to support product-line development [KMM05].

6.2.3 Evaluation of Process Automation and Tool Support

We used SODA to author a service-oriented model of our BART case study, as well as a model for an automotive Central Locking System (CLS). The tool enabled us to define and edit the core development artifacts: requirements, services, roles, interactions and components. We could successfully analyze the models for consistency, using a plug-in consistency checker that accessed the artifact object model. We could generate process supporting and system engineering documentation from the tool, including a requirements document and an architecture document.

The various described uses of the tools as part of an integrated tool landscape show that the process and its underlying artifact model can be well automated. Applying the powerful process and tools can make development projects very efficient.

[4] `http://autofocus.informatik.tu-muenchen.de/index-e.html/`
[5] `http://eclipse.org/aspectj/`

6.3 Qualitative Analysis Against Key Properties

In the following, we discuss how our solution addresses some of the issues with current development approaches listed in Section 1.2. There, we listed a number of key properties important for effective development approaches. In the following, we analyze how our approach addresses these properties.

Logical and abstract design models Our approach offers service-oriented model-based development that utilizes design models and model views on various levels of abstraction. The approach emphasizes in particular the development of a logical architecture and design model, supports iterative development and systematic model refinement. The system design model is founded on a precise metamodel; different model views across abstraction level are connected. Views of higher abstraction level include the Service View, Modal View, Role View and Interaction View.

Functional architecture System functions are expressed in our approach as services; as such they are part of the logical architecture and design model. Dependencies between services can be addressed directly within the interaction pattern specifications, for instance by applying High-Level MSCs and MSC composition operators. Alternatively, functional dependencies can be represented through the use of modes, roles and crosscutting property specifications. Interaction specifications are partial models; each interaction pattern defines its own set of behaviors. Only the combined set of interaction patterns yields the total system behavior.

Adequate design models Our solution supports the separation of independent concerns into distinct views on the system architecture and design model. Hierarchical system architecture, external component interfaces and services, internal component decomposition, operational modes, data entities, interaction patterns, deployment mappings and component configurations all have highly targeted and expressive views. We suggested flexible notations for these views; other notations are possible as long as they can be interpreted given our formal system model. Crosscutting properties can be expressed within multiple views. The underlying formal model ensures consistency between views, resulting in an integrated design model. The development process ensures consistency across development artifacts, supporting iterative and parallel project execution.

Formal system model We introduced a comprehensive formal system model suitable to express powerful properties supporting distributed reactive system design. The formal system model is based on a mathematical formalism and provides key definitions and operations required for flexible extension. It supports component-based specifications, distributed systems, hierarchical system architectures, composition and refinement, partial

service specifications and service combination. The formal system model provides the semantic domain for the precise interpretation of higher-level concepts, notations, and operations.

Domain independence Our approach supports the development of distributed, reactive systems and their architectures. It is not specific to any application domain. For instance, our approach supports the development of embedded automotive control systems as well as distributed information systems. Development strategies and processes may differ across application domains, given for instance the emphasis on real-time sensitive embedded systems vs. highly complex data-intensive information systems.

Seamless development Our development process addresses all phases of system development, from requirements engineering to system operations. It defines a comprehensive integrated artifact model providing artifacts for all levels of abstraction without leaving too large of a conceptual gap between levels. For instance, requirements can be expressed as use cases, refined and formalized into service interfaces, elaborated within interaction patterns, mapped to components, and integrated into component assemblies. The artifact model explicitly states artifact dependencies. The process defines activities to systematically increase the level of detail in system design, for instance through the creation of respective design model views and implementation artifacts. The discussed tool support assists seamless development efforts for instance by providing model transformations and model validations functions.

Iterative development Our development process supports iterative development through both constructive and analytic means. The process enables systematic model refinement and refactoring operations, and allows underspecification and nondeterminism supporting leaner models and easier design revisions. Our process offers analytic quality assurance activities, assessing multiple artifacts for consistency by evaluating product dependencies and consistency rules. Constructive and analytic means can be applied as needed and must be applied as required by the process, ensuring consistent artifact baselines throughout multiple project iterations.

Compositionality Components are a central concept in our development approach. A hierarchical system architecture of compositional components supports scalability. The explicit notion of component interfaces provides the required modularity. Components can be implemented, substituted, commissioned, acquired as COTS, verified and replaced flexibly as long as all their realizations respect the syntactic and behavioral component interfaces.

Service specifications and their refinements in form of interaction patterns provide a realization of the component interfaces. Services themselves are partial specifications that

offer operations suitable for modular development: service specifications can be developed in isolation—potentially reused—and combined when needed.

Maintainable design models System design model views provide somewhat independent views on certain topics of the system, such as external interface, high-level operational modes, data definitions and interactions. This makes them very helpful in reducing specification complexity and keeps them maintainable. At the same time, model views remain part of a larger integrated design model, ensuring model consistency. Changes in one model view cause changes or obligations for change in other model views. The system designer has the choice to temporarily permit inconsistent model views, in order to expedite larger design revisions. Tool support can perform changes across various model views automatically, keeping the designer focused on the work within one model view.

Tool support In a previous section, we demonstrated effective tool support for our approach. We introduced a comprehensive architecture for supporting the entire development process through a development tool. The SODA proof of concept implementation provides a platform for the integration of specialized artifact and model view editors and for model transformation tools; it spans the entire development process, provides project management support and can generate various forms of project documentation. We mentioned additional tools for rapid architecture evaluation, real-time monitoring of system properties, property verification and others as part of the tool landscape supporting our process and its underlying formal system model.

Characterized development approach The formal system model, process metamodel and process artifact model we introduced in previous chapters provide the basic foundations for a well characterized system development approach. This enables prospective users of this process to make clear determinations on whether the process is applicable to their project situation, and which benefits to expect when compared with a other approaches. The effort for the application of our process, including the development of all artifacts and the execution of all quality assurance activities can be estimated, leading to more repeatable project executions.

Development process We provided a system development process that embeds a powerful service-oriented development approach for distributed, reactive systems. The process supports in particular sophisticated views on the logical architecture and design model and enables iterative development, hierarchical system architectures and model refinement. We integrated our service-oriented development process with the V-Modell XT, enabling direct applicability in large-scale, long running project settings. Prospective process users can directly execute our process within the context of the V-Modell, after a straightforward process extension and tailoring step that can be supported by process tools.

Precise process definition Our process is based on a precisely defined process metamodel and on a comprehensive service-oriented artifact model, both realizing the foundations for a precise process definition. Artifacts are the main process outcomes. The artifact model defines these outcomes and their dependencies. Activities are directly linked to artifacts.

Summary Our solution comprised of service-oriented development approach and development process, based on a formal system model and process metamodel, satisfies all key properties listed above. The degree of support for each property is at least at a medium level and often high. Table 6.1 shows the degree of key property support for our approach. In the sections below, we qualitatively compare with the Catalysis component-based approach and the Rich Services approach.

Table 6.1: Our approach evaluated against success criteria

Property	**Degree of support with our approach**
Logical and abstract design models	++
Functional architecture	++
Adequate design models	++
Formal system model	++
Domain independence	+
Seamless development	+
Iterative development	+
Compositionality	++
Maintainable design models	+
Tool support	+
Characterized development approach	+
Development process	++
Precise process definition	++

6.4 Comparison with Existing Approaches

In this section, we compare our development approach with other approaches known from the literature or applied in practice. We discuss any significant commonalities and differences and briefly compare development approaches using the key properties introduced in Section 1.2.

6.4.1 Comparison with the Catalysis Approach

The *Catalysis* approach, introduced by D'Souza and Wills [DW98], is a software development method based on the UML language. As part of the UML ecosystem, it primarily focuses on the development of component-based software systems often applying object-oriented methods. We take it as an example of UML-based software development approaches.

One of the strengths of the Catalysis approach is its emphasis on design models as precise abstractions and formal specifications of system behavior. This goes beyond most other UML-based methods. The approach enhances use case modeling with precondition requirements and postcondition guarantees to precisely define the interpretation of a specification. The approach also uses UML's enhanced state and interaction models and emphasizes a general modularity of specifications, such that they can be reused in different contexts within the system design model or for different systems, for instance by specifying component "plug-in" interaction protocols.

Our approach shares many some common goals with Catalysis in that it is also model-based and targets mainly component-based systems. Both approaches encourage modular specifications and structured system architectures and include a choice of specification notations. Our approach particularly emphasizes services as higher level partial, precise model elements on an abstraction level in between requirements oriented use cases and detailed system component and state machine specifications.

The main difference is that Catalysis rests on UML and inherits its challenges with design model interpretation as discussed in Section 2.2.2. Our model is built on a formal mathematical system model and provides notations and methodology based on operations defined using the formal model. Other notations are possible as long as they can be mapped to the formal system model. Every element of a system specification thereby has a precise semantic interpretation and all views of the design model are related through an integrated metamodel, leaving no room for specification ambiguity. While it remains possible for the system designer to leave specifications temporarily inconsistent in order to expedite design cycles, the model requires to eventually return to a consistent baseline. The integrated artifact model strongly supports this. We complement this by providing an explicitly defined development process with artifacts and activities.

Table 6.2 shows the degree of support of development key properties with Catalysis.

6.4.2 Comparison with the Rich Services Approach

The Rich Services approach [DEF+07] is closely related to our service-oriented development process. This is no surprise given that both share common origins. Both the Rich Services approach [DEF+07] and an early variant [BFH+07] of the approach described in this thesis used the CoCoME (Common Component Modeling Example) case study [RRMP08] as

Table 6.2: Evaluation of other approaches against success criteria

Property	**Degree of support for Catalysis**	**Degree of support with Rich Services**
Logical and abstract design models	+	+
Functional architecture	○	++
Adequate design models	+	++
Formal system model	○	++
Domain independence	+	+
Seamless development	○	+
Iterative development	+	+
Compositionality	○	++
Maintainable design models	○	++
Tool support	++	+
Characterized development approach	○	+
Development process	○	+
Precise process definition	–	–

common modeling example, facilitating comparison. In the following, we summarize some of the commonalities and differences.

Both approaches are founded on a FOCUS based formal system model and include a formal semantics for the interpretation of MSC-based interaction specifications [Krü00b]. We have enhanced this formal model for our process with strong notions of hierarchical system architectures, leveraging component interfaces, interface abstractions and systematic use of refinement, see Section 3.3.3.

Both approaches support domain modeling [Eva03] as part of the process artifacts of the early system requirements analysis and specification phases. Figures 1.4 and 1.5 depict domain models formalizing the concepts surrounding the CoCoME system using the Rich Services approach [DEF+07]. These domain models can be used to inform high-level design entities such as Rich Services or component roles, respectively.

The main differences are in the design model views. Our approach supports the explicit specification of a hierarchical component-based system architecture with the system decomposed into distributed components, which can recursively decompose into subcomponents as shown in Figures 5.17 and 4.36. The Structure View in our logical architecture and design model specifies these decomposition relationships. Based on this system decomposition, components expose public interfaces (Interface View), subsequently refined by separate specifications of service interfaces (Service View), data entities (Data View), operational modes (Modal View). Both approaches support the definition of roles and service interaction patterns.

The Rich Services approach emphasizes composable Rich Services that abstract their interaction interfaces in form of Service-Data Connectors (SDCs). Figure 6.7 shows a Rich Service specification of the Cash Desk component of the CoCoME example. This specification references interaction patterns defined using MSC notations, similar to our interaction pattern specifications.

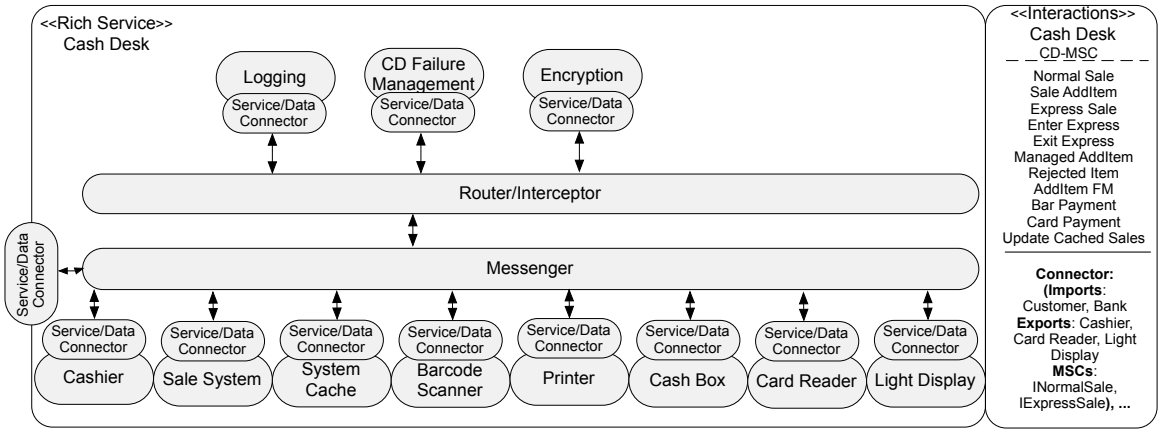

Figure 6.7: CoCoME Cash Desk rich service, from [DEF+07]

Rich Services provide an abstraction of message driven system integration environment, where services communicate according to their specified Service-Data Connectors, potentially intercepted by a router/messenger infrastructure. This infrastructure can be instructed to enforce infrastructure services to enforce crosscutting concerns. Figure 6.7 shows "Logging", "CD Failure Management" and "Encryption" infrastructure services. Other possible services include policy checking and identity management. In our approach, we model these crosscutting concerns explicitly as interactions with dedicated roles, e.g. a policy validation component or a failure manager, or as local actions performed by roles.

Both approaches provide a development process and associated tool support. Our approach goes beyond the current state of the Rich Services approach in that it defines a precise process documentation, comprising of process metamodel, artifact model and activity model. This supports the direct embedding of our approach into existing development processes, such as the V-Modell XT as described in Section 5.5.2 and in [MK07]. Significant extensions to the Rich Services approach exist [Men12, Dem13] that provide valuable methodology; we will discuss potential applications to our approach in the outlook for this thesis.

Table 6.2 shows the degree of support of development key properties with the Rich Services approach. As expected, Rich Services and our approach are rated similarly.

6.4.3 Comparison with Enterprise Architecture Frameworks

A class of enterprise architecture frameworks exist that attempt to organize architectural documentation resulting from all engineering processes within an enterprise without prescribing a certain methodology. They do this by providing a set of viewpoints or perspectives organizing all architectural documentation artifacts, and then further arranging them by stakeholder or target audience. The Zachman architecture framework [Zac06] has the furthest reaching origins, dating back to 1980. It categorizes perspectives by aspect of the architecture into "contextual", "conceptual", "logical", "physical" and "detailed" artifacts, and by the type of stakeholder, leading to a matrix of 30 perspectives. The framework also provides 36 categories for describing any complex entity and provides guidance for how to relate the different perspectives, e.g. in a top-down approach.

The Zachman architecture framework was specialized and adapted in numerous ways, and influenced enterprise frameworks with similar intent. National governments, for instance, defined their own architecture frameworks for the development and acquisition of large-scale systems, for instance as the U.S. Department of Defense Architecture Framework (DoDAF)[6] and the British MODAF[7]. DoDAF, for instance, provides 8 viewpoints, including the "operational", "systems", and "services" viewpoints. The services viewpoint, in particular, enables descriptions of performers, activities, services, and their interactions. It provides a place to describe solutions more abstract than the systems viewpoint, but not as business/mission oriented as the operational viewpoint.

Our process shares some intent with these enterprise architecture frameworks. We also provide an artifact model that covers a large part of the system engineering activities, and we categorize artifacts into requirements, logical design, technical design and implementation artifacts. In addition to providing an artifact model, our process provides methodology, and suggests notations and transformations. It presumes a specific system model and focuses on multifunctional, software-intensive, distributed reactive systems with mostly static component architectures, rather than providing a fully generic ontology of enterprise architecture. Similar to DoDAF 2.0, our process makes services first-class entities of the method, emphasizing their importance as a functional organization of system elements, less susceptible to change and more abstract than component or systems designs. DoDAF added the services perspective with its version 2.0, realizing the importance of this level of abstraction in system design. There is a correspondence between the viewpoints of DoDAF and our model views, for instance "SvcV-4 Services Functionality Description" has a similar purpose as our Service View. In a way, these architecture frameworks resemble the scope and intent of the V-Modell XT—albeit without the process part—and can benefit from extensions with a specific application of a service-oriented development method such as ours. Thereby all the benefits of the framework apply, enhanced with the benefits and tool automation support of a more specialized approach.

[6] DoDAF version 2.02, 2010, `http://dodcio.defense.gov/Library/DoDArchitectureFramework.aspx`
[7] MODAF, `https://www.gov.uk/guidance/mod-architecture-framework`

6.5 Applying Service-Oriented Concepts in Practice

We worked on applying elements of our service-oriented development process in practice with the architecture of the U.S. National Science Foundation's Ocean Observatories Initiative Cyberinfrastructure (OOI CI) [Ini08, SGO+10, CAF+09]. The resulting system, *OOINet* [OOI14], was designed using service-oriented concepts and provides a service-oriented implementation. The Scientific Observatory Network (SciON) [Sci15] is a recent open source software release of the core components of the OOINet software. The applied development process does not fully match our described process but still offers insight into valuable lessons learned from practical service engineering:

- *Domain analysis:* For a complex system or application domain, we found it very valuable to analyze the concepts of the application domain and their dependencies, and to document these in form of domain models. The effort it takes to understand the concepts and to bring them into a consistent representation is well spent, in order to avoid costly revisions of requirements, redesigns of the service or component architecture model, and communication inefficiencies due to a lacking agreed on domain vocabulary. Domain models can be readily used to extract requirements, use cases, system functions and services and glossaries of terms to be used in the early phases of a development project. In particular, it becomes more straightforward to establish a sound service architecture based on a comprehensive domain model.
- *Architect availability and communication:* In a larger development project spanning multiple releases, it is very important to document the core architecture and its underlying principles and to maintain a living documentation of the architecture. It is equally important that architects who fully understand the system with its stakeholders and constraints communicate that architecture regularly and are available in management and development meetings to answer domain and technical questions. When necessary, they need to defend an existing architecture and previously made design decisions against changes of requirements, changes to of core concepts and sloppy implementations. This is particularly true for a complex distributed system architecture, where each service only contributes a piece to the overall system and the weakest link, e.g. a badly implemented service, can affect the entire system.
- *Developer training and acceptance of designs:* Designing a system using services and implementing service components requires developers trained in service-oriented architectures. It is important to staff a project with developers having this capability set, in particular for systems with many distributed components, in order to avoid system quality issues and development delays. Developers need to understand service architectures and dependency chains, and the consequences of design changes on system security, stability and performance. Developers need to fully understand service-oriented designs received from architects.
- *Concurrent design and development:* Developers can work on many services and components of a service-oriented system in parallel. This is very beneficial in general

and developers can work independently. A big issue are dependencies and system integration. If one service depends on a service or infrastructure component that is concurrently in development and gets delivered initially in limited or partially correct version, then this can cause significant inefficiencies to development progress. Changes to the implementation of a component can cause major delays and "ripples" to dependent components and already tested functionality. It can take several iterations to resolve such issues. The basic answer is of course to develop against agreed on mutual interfaces, but often these interfaces need to change as a result of lessons learned during development. A few technically impractical designs discovered during implementation can require elaborates changes to the system design. A viable strategy is to limit dependency chains initially and to design simple versions of services and components early, before enhancing them subsequently.

- *Services work:* We found that a large-scale service oriented architecture helps to address system complexity by simplifying design and implementation effort of the system's distributed elements. Instead of grand designing and adding on to complex components, it becomes possible to work independently on modular, smaller services that can be combined into more complicated functionalities as the system grows. Service-oriented architectures scale well and services can be modified independently. As discussed, this requires strong emphasis on a sound service architecture and a thorough domain analysis, as well as risk mitigation activities to limit the consequences of unforeseen design changes in a highly interdependent system architecture.

6.6 Shortcomings

Every development approach realizes a trade-off providing a targeted solution for a specific class of systems or a specific class of development projects. More flexible approaches attempt to provide a framework that can be extended and tailored to increase applicability to a variety of situations. Our approach combines a sophisticated conceptual model with a flexible process framework. It is not limited to one application domain as explained above and it supports a variety of project execution variants. Nonetheless, our approach realizes specific trade-offs that might not suit all situations. The following is a list of known shortcomings:

Complexity Our approach targets the development of larger-scale distributed reactive systems. Addressing some of the complexities of this system class, such as interaction complexity in distributed environments and intertwined functional dependencies, required the introduction of powerful abstractions, modeling notations and a comprehensive underlying formal system model. All concepts need to be understood by system designers. Designers must be proficient in applying good design patterns to maintain a flexible design model. This of course implies that the project needs upfront access to such highly trained

individuals. Some notations may look intuitive at first glance, but may come with conceptual "strings attached". For instance, some refinements may not be possible in certain situations, where the existence of one interaction pattern, operational mode or role may preclude a desired model change. The systematic development process alleviates some of these concerns by offering method and tool guidance and automation through tool support.

Effort The effort to keep process artifacts consistent and to maintain a consistent logical architecture and design model is relatively high. This is reasonable for larger-scale development projects where the costs of errors and specification inconsistencies far outweigh the increased upfront process overhead. Process automation and tool support help to reduce some of the process overhead.

Lack of agility A system design model with multiple model views and tracing links to other process artifacts is naturally less agile than a development method such as "Extreme Programming". If a domain is well understood, a well-trained or small team is available, the system size is limited and rapid prototyping results are required, it may be better to apply a lighter weight process. This is risky though, because often such first prototypes long exceed their expected life span and then need to be maintained without proper design models, consistent application of architectural principles and doubtful nonfunctional properties such as security. At some point, an agile project becomes less agile and then needs to be maintained in a less than optimal situation.

Partial specification complexity Working with partial specifications has strong benefits. New interaction patterns can be added to model new use cases and additional component behavior. This flexibility has a downside. It becomes very difficult for the system designer to fully comprehend a component's behavior and resulting state space. Given that the full component behavior model only gets derived automatically as a result of a combination of all partial specifications, the resulting behavior state model is often not understandable by humans and cannot easily be optimized. It may require multiple feedback cycles of changing some of the higher level design model views and regenerating the component state model in order to accomplish a desired result. On the other hand, manually understanding a component's state space and trying to optimize it is generally not a good idea because it is very complex to "get it right" and keep it so.

Need for tool support Given the described model complexity and process overhead, having process automation available through tool support is basically a requirement. We introduced a tool support architecture and demonstrated its feasibility through a proof of concept implementation and powerful existing tools that can be integrated. Completing this tool landscape and providing a polished, intuitive, effective platform for end users to use remains future work with substantial associated effort, given the sophistication of the underlying process and design model views.

6.7 Summary

In this chapter, we have evaluated our service-oriented development approach using a combination of evaluation strategies. We introduced a viable tool architecture to demonstrate the practical applicability of our approach. Existing proof of concept implementations and powerful tools demonstrate that comprehensive tool support is feasible and can be implemented, supporting the entire development process from requirements engineering to system operations, and supporting the elements of our process artifact model, including all design model views. Tool support and process automation are major steps towards practical applicability of an approach as part of a comprehensive engineering solution.

Subsequently, we compared our approach against a list of key properties of development approaches that make them effective for always increasing system scale and system complexity, while helping to increase result quality and keep development cost at bay. Our process addresses all properties with at least medium to high degree of support. We also compared our approach against Catalysis, an UML-based development method for component-based systems, and against Rich Services, an approach closely related to ours leveraging service interaction specifications, and mentioned commonalities with enterprise architecture frameworks. Every approach has different strengths and shortcomings. By comparing using the key properties mentioned above, we found that our approach ranks equal or higher for most properties, providing a comprehensive solution to prospective process users.

We presented lessons learned from applying select elements of our process and artifact model in practice, indicating a positive benefit of applying service-oriented concepts to large-scale system development. We listed known shortcomings of our approach to enable prospective users to determine situations when not to apply our approach, and to point to future required work.

7 Summary and Outlook

This chapter summarizes the work presented in this thesis, draws a conclusion and provides an outlook to further work and research opportunities.

Contents

7.1 Summary

In this thesis, we have introduced, evaluated and discussed a service-oriented process for the development of distributed reactive systems, focusing on the development of logical system architectures and design models. Core constituent elements of our approach are an iterative development process based on a thorough artifact model, and a powerful service-oriented development technique.

We motivated this approach in the introduction by explaining current challenges in distributed system development. We introduced a list of key properties that development approaches need to satisfy in order to remain effective for the development of increasingly larger-scale distributed reactive systems, delivering high quality at bounded cost. Chapter 2 introduced service-orientation as promising strategy to addressing present challenges with the development of distributed systems and documented its benefits to all phases of system development. We explained the importance of development processes for a successful, repeatable execution of software development projects.

As a solution to addressing the stated challenges, we introduced a systematic development process applying model-based and service-oriented development techniques. First, we described an existing comprehensive formal model of distributed systems suitable to express the key properties of distributed reactive systems, in particular the interaction behavior of distributed system components. This formal model in Chapter 3 provides a strong foundation for subsequent definitions of all elements of our approach and a semantics domain for the interpretation of system design models. It is a consistent presentation of existing formal theories and definitions.

The main contribution of our work started with Chapter 4, introducing an artifact model for a model-based service-oriented development process, suitable to represent the concepts of our formal model. We provided domain models describing all relevant system development artifacts and their dependencies. The logical system architecture and design model is central to all artifacts; it represents an integrated, metamodel-based design model providing a number of interconnected views on the model elements. We suggested notations for these views and provided formal definitions for model views and notations utilizing the previously introduced formal system model. Services realize first-class entities throughout the development process in our artifact model and are part of many design model artifacts. We illustrated artifacts and notations by providing examples using our running example, the BART traffic control system.

A system development process in Chapter 5 puts it all together, combining work products from the artifact model with activity flows and descriptions. The process supports iterative development and seamless transitions between all development activities. We integrated this process with the V-Modell XT, a well established system engineering process framework, making our solution readily available to projects in need for a flexible, scalable development process and looking for repeatable development results.

To evaluate our solution in Chapter 6, we showed how to automate central parts of our process by providing an architecture for a comprehensive tool landscape, centered around an extensible process support platform. We named powerful existing tools for code generation, architecture exploration and runtime monitoring. We described SODA, a proof of concept implementation that we created for the process tool platform and how it simplifies development. Results indicate that automation is feasible and effective in supporting the process seamlessly and iteratively. We analyzed how our process satisfies the key properties for development approaches introduced in the motivation. Our process addresses all of them and provides a substantial advancement over existing solutions. We compared our approach to two existing development approaches. We listed known shortcomings found during the evaluation and described how to address or avoid them.

7.2 Conclusion

We conclude that we have introduced an effective development process, addressing some of the most pressing challenges in distributed system development. Service-orientation as a concept to advance system design can be successful in managing the ever increasing complexities of modern systems, while increasing result quality and limiting development cost. The management of functional dependencies and interaction complexity of system designs, in particular, offers tangible benefits and justifies the positioning of services as first-class entities throughout the development process. A systematic development process provides repeatable development performance at quantifiable cost. A service-oriented V-Modell XT extension offers a practice proven process that can scale to the largest development projects and system sizes.

7.3 Outlook and Future Work

The service-oriented process we have introduced in Chapters 4 and 5 provides a comprehensive development approach for distributed systems. Given the breadth of the presented material covering several main system engineering activities, additional work is required or suggested to fully flesh out the approach.

In particular, the following topics of additional research seem promising:

- refine the development process with details,
- enhance the formal model and show further applications,
- provide domain specific adaptations and case studies,
- improve tool support.

The sections below describe possible work in more detail.

Development Process Our process artifact model provides a domain model for the core development artifacts and model elements. When describing the artifacts and related process activities, we focused on the system design phase and kept the description of process elements of other phases rather cursory. Requirements artifacts and activities, in particular, can benefit from greater detail descriptions. There exists substantial research on the systematic elicitation and management of requirements in a model-driven development process, cf. [Gei05, HRSW06, Deu08, Rit08, Pen11, Vog15], with the potential of seamlessly integrating with a design-focused approach such as ours. The work of Penzenstadler [Pen11] on the derivation of subsystem requirements from system requirements can be directly related to our hierarchical system architecture, enabling an iterative refinement of requirements and architecture model with a tight tracing between both artifacts. The work of Vogelsang [Vog15] can provide the necessary methodology to manage feature interactions in multifunctional systems on a level of requirements that can subsequently be related to the service architecture of our approach, in particular evaluating the consistency of service refinement and service relationships in our design model.

Document templates and example documents are highly useful for the practical application of development processes, cf. the materials provided for the V-Modell XT [Bun12] and the Rational Unified Process [Kru00a]. Such templates—also called product templates—could be available for download or included within a process tailoring tool in order to generate project specific document templates. In particular the initial creation of design models with their many model views, and their linkage to document artifacts may benefit from automation, existing templates and additional sets of examples.

Formal Model We have presented a comprehensive formal model sufficient to serve as semantics domain for the definition of system architectures, distributed components and various forms of their specification. The formal model consistently integrates various existing theories and allows reasoning about the effects of model transformations such as refinement and refactoring steps, and the precise definition of the meaning of various design model notations. The formal model particularly covers the meaning of distributed system interactions and specification formats such as our MSC-based service models. We did not include formalizations of requirements engineering steps and of more technical integration, deployment and operations activities. Given the existing research on formal requirements engineering, cf. [Gei05, HRSW06, Rit08, Pen11, Vog15], in some parts directly based on the FOCUS theory, it is possible to extend the formal model consistently with additional definitions and model transformations covering the early phases of system development.

The formal model would also benefit from additional specialized refinement and refactoring transformations. This is also where additional examples would be useful to demonstrate the value of the formal model to higher level questions of tool support and day to day development.

In [AKMP05] we showed how to use design specifications of distributed system components

to perform runtime monitoring of the implemented and deployed components within a distributed system architecture. In this example, we applied monitoring at the level of a specific inter-process communication (IPC) infrastructure, in particular observing message flows and timing deadlines. In [KMM06] we showed how to use aspect-oriented techniques to generate or augment component implementations. Additional work could explore other forms of runtime monitoring and the generation of monitors, for instance wrapping deployed service components with message interceptors, cf. our collaboration with [BCD+13].

Our collaboration work published in [SAB+13] focused on the formal representation of domain-relevant relationships of distributed components, such as agreements resulting from a successful negotiation conversation, and their subsequent monitoring and enforcement. The knowledge expressed in these formal models could be added to the Data View of our service oriented model, and subsequently be used during runtime to monitor the correct behavior of implemented services. A formalization using our FOCUS-based formal model could be attempted to achieve a more comprehensive formal model.

A substantial and highly valuable effort would be the generalization and extension of the formal model and respective elements of the notations and process to dynamic system architectures cf. [Bro14, Sal02], such as modern cloud-based infrastructures with dynamically extending and shrinking footprints based on user load. Similarly beneficial could be the incorporation of the particular properties of mobile applications, such as unreliable networks, limited resources, and the need to manage connections and state for a vast number of mobile clients.

Domain Specific Adaptation and Case Studies We designed our approach to be domain-independent and of general applicability for the development of distributed system architectures. A lot of detail and technology context exists within the various application domains. Future work could explore some of these application domains in detail, say Internet-based systems applying a Microservices [New15] style architecture, with a distributed service-oriented backend and mobile clients. This could be done as case studies or by reverse-engineering existing process artifacts of completed development projects. Studies and documented findings investigating operational, real-world systems of larger scale would be particularly instructive.

A valuable secondary outcome of such application domain specific case studies could be tailored specializations or variants of our service-oriented development process, potentially even packaged as V-Modell XT extensions, contributing domain specific knowledge and adaptations to practitioners who are ready to start a development project. Method guidance and tool suggestions are of particular value, including references to best practices, example projects, useful development environments, suggested candidate architectures and lessons learned from the field. Modular extensions comprising notations, process elements and enhanced tooling could be defined. For instance there exists a plethora of implementation level technologies for Web enabled service-oriented architectures. Guidance on business process modeling, REST Web APIs, WS-* Web Service standards, architectural

patterns, and an integration with service-oriented middleware such as Enterprise Service Buses can be part of a comprehensive specialized variant of our process. The availability of such concrete process documentation may increase the rating of our approach for the key properties "Characterized development approach" and "Tool support", cf. 6.3, increasing immediate applicability.

An interesting case study could be investigating the applicability of Rich Services, see Section 6.4.2, within the larger umbrella of our service-oriented development process. Menarini [Men12] for instance introduces an aspect-oriented approach for the composition of Rich Services. Demchak [Dem13] applies policy-driven development (PDD) in order to inject workflows into systems using Rich Services. These approaches make use of the fact that partial interaction specifications are valuable in specifying specific or conditional system behavior, for instance the services of a system enabled or disabled for product line variants. A systematic method is required to specifying a system comprised of the various applicably partial specifications, and a constructive approach to generating the resulting system of deployable components. Rich Services introduce a powerful notion of hierarchy and composition. The case study could compare our version of specifying systems composed of multiple services, then subsequently performing a closed-world assumption and component synthesis, with the cited ways of composing and injecting Rich Services.

Tool Support Good and automated tooling is essential for the adoption of a development approach, as stated in the evaluation. Future work should advance the development of an end-to-end prototype for a comprehensive process automation tool environment, potentially building on a modern integrated development environment such as Eclipse[1] with its process framework EPF[2]. There already exist a significant number of plug-ins for process support and model-based development[3] that can be leveraged or integrated in a comprehensive tool suite.

Of particular value are extensions of the tooling targeting specific application domains, technology sets and architecture styles, as mentioned above. A flexible, extensible tool architecture, such as the one sketched in Section 6.2.2 and implemented effectively, will be a basis for integrating various tools, extensions and plug-ins. These contributions could result from case studies or be made by the community in support of our service-oriented approach. Providing an extensible tooling ecosystem and a marketplace to find the right tool for the right project will go a long way towards broad acceptance of a development approach, which is of course largely dependent on the inclination of the developer community.

[1] `https://eclipse.org/`
[2] `https://eclipse.org/epf/`
[3] E.g. `http://marketplace.eclipse.org/content/togaf-designer-juno-version`, `http://marketplace.eclipse.org/content/spem-designer-indigo-version`

Bibliography

[ACF+09] Matthew Arrott, Alan D Chave, Claudiu Farcas, Emilia Farcas, Jack E Kleinert, Ingolf Krueger, Michael Meisinger, John A Orcutt, Cheryl Peach, Oscar Schofield, et al. *Integrating marine observatories into a system-of-systems: Messaging in the US Ocean Observatories Initiative*. IEEE, 2009.

[AF95] DE Avison and G. Fitzgerald. *Information systems development: methodologies, techniques and tools*. Paul & Company Publishers Consortium, Inc., 1995.

[AG96] Robert Allen and David Garlan. The wright architectural specification language. *Rapport technique CMU-CS-96-TBD, Carnegie Mellon University, School of Computer Science*, 1996.

[AG07] João Paulo A Almeida and Giancarlo Guizzardi. On the foundation for roles in rm-odp: contributions from conceptual modelling. In *EDOC Conference Workshop, 2007. EDOC'07. Eleventh International IEEE*, pages 205–215. IEEE, 2007.

[AGA+10] A. Arsanjani, S. Ghosh, A. Allam, T. Abdollah, S. Ganapathy, and K. Holley. SOMA: A method for developing service-oriented solutions. *IBM Systems Journal*, 47(3):377–396, 2010.

[AH06] Paul Allen and Sam Higgins. *Service Orientation: Winning Strategies and Best Practices*. Cambridge University Press, 2006.

[AKMP05] Jaswinder Ahluwalia, Ingolf Krüger, Michael Meisinger, and Walter Phillips. Model-Based Run-Time Monitoring of End-to-End Deadlines. In *Proceedings of the Conference on Embedded Systems Software (EMSOFT)*, 2005.

[Amb99] Scott W. Ambler. *More Process Patterns: Delivering Large-Scale Systems Using Object Technology*. Cambridge University Press, 1999.

[And08] R.J. Anderson. *Security Engineering: A guide to building dependable distributed systems*. Wiley, 2008.

[Arc00] Architecture Working Group of the Software Engineering Committee and others. Recommended Practice for Architectural Description of Software Intensive Systems (IEEE 1471-2000). *IEEE Standards Department*, 2000.

[Art06] David J.N. Artus. SOA realization: Service design principles. Technical report, IBM developerWorks, 2006. Version of 17-Feb-2006, `http://www-128.ibm.com/developerworks/webservices/library/ws-soa-design/`.

[Aut06a] AutoFocus. Website, 2006. `http://autofocus.informatik.tu-muenchen.de/index-e.html`.

[Aut06b] AutoSAR. Website, 2006. `http://www.autosar.org`.

[BA00] K. Beck and C. Andres. *Extreme programming explained*. Addison-Wesley Reading, MA, 2000.

[Bar06] Barry & Associates, Inc. Web Services and Service-Oriented Architectures. Website, 15-Nov 2006. `http://www.service-architecture.com/`.

[BB08] Robert Battle and Edward Benson. Bridging the semantic web and web 2.0 with representational state transfer (rest). *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(1):61–69, 2008.

[BBR+07] Andreas Bauer, Manfred Broy, Jan Romberg, Bernhard Schätz, Peter Braun, Ulrich Freund, Nuria Mata, Robert Sandner, Pierre Mai, and Dirk Ziegenbein. Das AutoMoDe-Projekt: Modellbasierte Entwicklung softwareintensiver Systeme im Automobil. *Informatik - Forschung und Entwicklung*, 2007. Accepted for publication.

[BCD+13] Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. In *Formal Techniques for Distributed Systems*, pages 50–65. Springer, 2013.

[BCHG06] P. Bocij, D. Chaffey, S. Hickie, and A. Greasley. *Business information systems: technology, development and management for the e-business*. Pearson Education, 2006.

[BCK03] L. Bass, P. Clements, and R Kazman. *Software Architecture in Practice*. Addison-Wesley, 2nd edition, 2003.

[BCSS99] Guruduth Banavar, Tushar Chandra, Robert Strom, and Daniel Sturman. A case for message oriented middleware. In *Distributed Computing*, pages 1–17. Springer, 1999.

[BD87] S. Budkowski and P. Dembinski. An introduction to Estelle: a specification language for distributed systems. *Comput. Netw. ISDN Syst.*, 14(1):3–23, 1987.

[BDD+92] Manfred Broy, Frank Dederichs, Claus Dendorfer, Max Fuchs, Thomas F. Gritzner, and Rainer Weber. The Design of Distributed Systems. An Introduction to FOCUS – Revised Version –. Technical Report TUM-I9202-2, Technische Universität München, 1992.

[BDG+06] Manfred Broy, Norbert Diernhofer, Johannes Grünbauer, Michael Meisinger, Martin Rappl, Sabine Rittmann, Bernhard Schätz, Maurice Schoenmakers, and Bernd Spanfelner. Service-Oriented Development - Whitepaper. Technical report, Lehrstuhl für Software & Systems Engineering, Technische Universität München, 2006.

[BFH+07] Manfred Broy, Jorge Fox, Florian Hölzl, Dagmar Koss, Marco Kuhrmann, Michael Meisinger, Birgit Penzenstadler, Sabine Rittmann, Bernhard Schätz, Maria Spichkova, and Doris Wild. Modeling CoCoME with Focus/AutoFocus. In *The Common Component Modeling Example. Comparing Software Component Models*, volume 5153 of *LNCS*. Springer, 2007.

[BG92] Gerard Berry and Georges Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BHB+03] Gerd Beneken, Ulrike Hammerschall, Manfred Broy, Maria Victoria Cengarle, Jan Jürjens, Bernhard Rumpe, and Maurce Schoenmakers. Componentware - State of the Art 2003. In *Proceedings of the CUE Workshop Venedig*, 2003.

[BK04] Manfred Broy and Ingolf Heiko Krüger, editors. *Pre-Proceedings of the Automotive Software Workshop San Diego 2004*. UCSD, 2004. `http://aswsd.ucsd.edu/2004`.

[BKM06] Manfred Broy, Ingolf Heiko Krüger, and Michael Meisinger, editors. *Automotive Software - Connected Services in Mobile Networks. Proceedings of the Automotive Software Workshop San Diego 2004*. Lecture Notes in Computer Science, Volume 4147, Springer, New York, 2006.

[BKM07] Manfred Broy, Ingolf Krüger, and Michael Meisinger. A Formal Model of Services. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(1), 2007.

[BKM08] Manfred Broy, Ingolf H. Krüger, and Michael Meisinger, editors. *Model-Driven Development of Reliable Automotive Services*. Number 4922 in Lecture Notes in Computer Science. Springer, Heidelberg, 2008. available at http://www.springer.com/computer/communications/book/978-3-540-70929-9.

[BKT+06] K. Balasubramanian, A.S. Krishna, E. Turkay, J. Balasubramanian, J. Parsons, A. Gokhale, and D. Schmidt. Applying model-driven development to distributed real-time and embedded avionics systems. *International Journal of Embedded Systems*, 2(3):142–155, 2006.

[BLHL+01] T. Berners-Lee, J. Hendler, O. Lassila, et al. The Semantic Web. *Scientific American*, 284(5):28–37, 2001.

[Blo97] Johan Blom. Formalisation of Requirements with Emphasis on Feature Interaction Detection, 1997.

[Boe88] Barry W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[Boo93] Grady Booch. *Object-oriented analysis and design with applications*. Addison Wesley Longman Publishing Co., Redwood City, CA, 1993.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, 2007.

[Bro87] Fred P. Brooks. No Silver Bullet – Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, 1987.

[Bro97] Manfred Broy. Requirements Engineering for Embedded Systems. In *Proceedings of FemSys'97*, 1997.

[Bro98] Manfred Broy. Compositional Refinement of Interactive Systems Modelled by Relations. *Compositionality: The Significant Difference*, pages 130–149, 1998.

[Bro03a] Manfred Broy. *Object-oriented programming and software development: a critical assessment*. Springer-Verlag, 2003.

[Bro03b] Manfred Broy. Service-Oriented Systems Engineering: Modeling Services and Layered Architectures. In *Hartmut König, Monika Heiner, Adam Wolisz. Formal Techniques for Networked and Distributed Systems - FORTE 2003*, pages 48–61. Springer-Verlag Heidelberg, 2003.

[Bro05a] Manfred Broy. A semantic and methodological essence of message sequence charts. *Sci. Comput. Program.*, 54(2-3):213–256, 2005.

[Bro05b] 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: Proceedings of the NATO Advanced Study Institute*, pages 47–82. Springer, 2005.

[Bro05c] Manfred Broy. The Impact of Models in Software Development. In *Lecture Notes in Computer Science, Volume 2605*, pages 396–406. Springer Verlag, 2005.

[Bro06] Manfred Broy. A Theory of System Interaction: Components, Interfaces, and Services. In Armin B. Cremers, Rainer Manthey, Peter Martini, and Volker Steinhage, editors, *Interactive Computation*, pages 41–96. Springer Berlin Heidelberg, 2006.

[Bro07a] Manfred Broy. Interaction and Realizability. In *Proceedings of the 33rd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2007)*, number 4362 in Lecture Notes in Computer Science, pages 29–50. Springer Berlin / Heidelberg, 2007.

[Bro07b] Manfred Broy. Two Sides of Structuring Multi-Functional Software Systems: Function Hierarchy and Component Architecture. In *Proceedings of the 5th ACIS International Conference on Software Engineering Research, Management & Applications (SERA 2007)*, pages 3–12. IEEE Computer Society Washington, DC, USA, 2007.

[Bro10] Manfred Broy. Multifunctional software systems: Structured modeling and specification of functional requirements. *Sci. Comput. Program.*, 75(12):1193–1214, 2010.

[Bro14] Manfred Broy. A model of dynamic systems. In *From Programs to Systems. The Systems perspective in Computing*, pages 39–53. Springer, 2014.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces and Refinement*. Springer New York, 2001. ISBN 0-387-95073-7.

[BSD03] B. Benatallah, QZ Sheng, and M. Dumas. The Self-Serv environment for Web services composition. *Internet Computing, IEEE*, 7(1):40–48, 2003.

[Bun12] Bundesrepublik Deutschland. *V-Modell XT*, 1.4 edition, 2012. `http://www.v-modell-xt.de/`.

[BvRS02] Peter Braun, Michael von der Beeck, Martin Rappl, and Christian Schröder. Automotive Software Development: A Model-Based Approach. *In-Vehicle Software*, 2002.

[CAF+09] A.D. Chave, M. Arrott, C. Farcas, E. Farcas, I. Krueger, M. Meisinger, J.A. Orcutt, F.L. Vernon, C. Peach, O. Schofield, and J.E. Kleinert. Cyberinfrastructure for the US Ocean Observatories Initiative: Enabling Interactive Observation in the Ocean. In *Proceedings of the OCEANS '09 IEEE Conference*. IEEE, 2009.

[CBB+10] Paul Clements, Felix Bachmann, Len Bass, David Garlan, Paulo Merson, James Ivers, Reed Little, and Robert Nord. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2010.

[CDK05] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems: concepts and design*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 4th edition, 2005.

[CES83] Edmund Melson Clarke, E. Allen Emerson, and Aravinda Prasad Sistla. *Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach*. ACM Press New York, NY, USA, 1983.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design. A Foundation*. Addison Wesley, 1988.

[CMM06] Capability Maturity Model Integration (CMMI), version 1.2. Software Engineering Institute of Carnegie Mellon University, 2006. `http://www.sei.cmu.edu/cmmi/`.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and JA Plaice. LUSTRE: a declarative language for real-time programming. *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188, 1987.

[CZST05] Artur Caetano, Marielba Zacarias, Antonio Rito Silva, and Jose Tribolet. A role-based framework for business process modeling. In *System Sciences, 2005. HICSS'05. Proceedings of the 38th Annual Hawaii International Conference on*. IEEE, 2005.

[Dan03] Daniel E. Atkins and Kelvin K. Droegemeier and Stuart I. Feldman and Hector Garcia-Molina and Michael L. Klein and David G. Messerschmitt and Paul Messina and Jeremiah P. Ostriker and Margaret H. Wright. Revolutionizing science and engineering through cyberinfrastructure. In *Report of the National Science Foundation Blue-Ribbon Advisory Panel on Cyberinfrastructure*. Springer, 2003.

[Day08] John Day. *Patterns in Network Aarchitecture: A Return to Fundamentals*. Prentice Hall, 2008.

[dBLK+05] J. de Bruijn, H. Lausen, R. Krummenacher, A. Polleres, L. Predoiu, M. Kifer, and D. Fensel. The Web Service Modeling Language WSML, 2005. WSML Final Draft 5 October 2005.

[DDH72] OJ Dahl, EW Dijkstra, and CAR Hoare. *Structured Programming*. Academic Press Ltd. London, UK, UK, 1972.

[DEF+07] B. Demchak, V. Ermagan, E. Farcas, T.J. Huang, I.H. Krüger, and M. Menarini. A Rich Services Approach to CoCoME. In *Lecture Notes In Computer Science*, pages 85–115. Springer-Verlag Berlin, Heidelberg, 2007.

[Dem13] Barry Demchak. *Policy Driven Development: SOA Evolvability through Late Binding*. Ph.d. dissertation, University of California, San Diego, 2013.

[Deu08] Martin Roland Deubler. *Strukturierte Nutzungssicht fr multifunktionale Systeme*. Dissertation, Technische Universität München, 2008.

[DGH+05] Martin Deubler, Johannes Grünbauer, Andreas Holzbach, Gerhard Popp, and Guido Wimmel. Kontextadaptivität in dienstbasierten Softwaresystemen. Technical Report TUM-I0511, Technische Universität München, July 2005.

[DGMR03a] Martin Deubler, Michael Gnatz, Michael Meisinger, and Andreas Rausch. Anforderungsanalyse für das V-Modell 200x. Projekt WEIT Ergebnisdokument, 8-May-2003, Lehrstuhl für Software & Systems Engineering, Technische Universität München, 2003.

[DGMR03b] Martin Deubler, Michael Gnatz, Michael Meisinger, and Andreas Rausch. Grobstruktur für das V-Modell 200x. Projekt WEIT Ergebnisdokument, 12-May-2003, Lehrstuhl für Software & Systems Engineering, Technische Universität München, 2003.

[DGP+04] Martin Deubler, Johannes Grünbauer, Gerhard Popp, Guido Wimmel, and Christian Salzmann. Towards a Model-Based and Incremental Development

Process for Service-Based Systems. In *Proceedings of the IASTED International Conference on Software Engineering (IASTED SE 2004), Innsbruck*, 2004.

[DH99] F. Dietrich and J. Hubaux. Formal methods for communication services. In *Technical Report No. SSC/1999/023, Institute for computer Communications and Applications (ICA), EPFL Lausanne, Switzerland.*, 1999.

[DH01] Werner Damm and David Harel. Lscs: Breathing life into message sequence charts. *Formal methods in system design*, 19(1):45–80, 2001.

[Dij68] E.W. Dijkstra. Letters to the editor: Go To Statement Considered Harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[Dij70] Edsger W. Dijkstra. *Notes on Structured Programming.* Technological University, Dept. of Mathematics, 1970.

[DKMR05] Martin Deubler, Ingolf Krüger, Michael Meisinger, and Sabine Rittmann. Modeling Crosscutting Services with UML Sequence Diagrams. In *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, 2005.

[DvLF93] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed Requirements Acquisition. *Selected Papers of the 6th Intl. Workshop on Software Specification and Design*, pages 3–50, 1993.

[DW98] Desmond D'Souza and Alan Cameron Wills. *Objects, Components, and Frameworks with UML: The Catalysis(sm) Approach.* Addison-Wesley Professional Reading, 1998.

[EHK+07] Vina Ermagan, To-Ju Huang, Ingolf Krüger, Michael Meisinger, Massimiliano Menarini, and Praveen Moorthy. Towards Tool Support for Service-Oriented Development of Embedded Automotive Systems. In *Proceedings of the Dagstuhl Workshop on Model-Based Development of Embedded Systems (MBEES'07), Technical Report 2007-01.* Technische Universität Braunschweig, 2007.

[EHS98] Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. *SDL. Formal Object-oriented Language for Communicating Systems.* Prentice Hall, 1998.

[Erl04] Thomas Erl. *Service-oriented architecture.* Prentice Hall Englewood Cliffs, 2004.

[Est04] SCADE Language Reference Manual. Esterel Technologies, 2004.

[Eva03] Eric Evans. *Domain Driven Design.* Addison-Wesley, 2003.

[Fah05] Michael Fahrmair. *Kalibrierbare Kontextadaption für Ubiquitous Computing.* Dissertation, Technische Universität München, 2005.

[FB02] D. Fensel and C. Bussler. The Web Service Modeling Framework WSMF. *Electronic Commerce Research and Applications*, 1(2):113–137, 2002.

[FG98] Jacques Ferber and Olivier Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. In *Multi Agent Systems, 1998. Proceedings. International Conference on*, pages 128–135. IEEE, 1998.

[Fie99] Roy Fielding. Hypertext Transfer Protocol - HTTP/1.1. Internet Engineering Task Force RFC 2616, June 1999. `http://www.ietf.org/rfc/rfc2616.txt`.

[Fie00] Roy Fielding. Representational state transfer. *Architectural Styles and the Design of Netowork-based Software Architecture*, pages 76–85, 2000.

[FK04] I. Foster and C. Kesselman. *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann, 2004.

[Fos05] Ian Foster. Service-Oriented Science. *Science*, 308(5723):814–817, 2005.

[Fow99] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[Fre04] R.L. Freeman. *Telecommunication system engineering*. Wiley-IEEE Press, 2004.

[Gar03] Lars Marius Garshol. BNF and EBNF: What are they and how do they work. *acedida pela última vez em*, 16, 2003.

[GDMR04] Michael Gnatz, Martin Deubler, Michael Meisinger, and Andreas Rausch. Towards an Integration of Process Modeling and Project Planning. In *Proceedings of the 5th International Workshop on Software Process Simulation and Modeling (ProSim 2004). ICSE 2004*. ACM Press, 2004.

[Gei05] Eva Geisberger. *Requirements Engineering eingebetteter Systeme – ein interdisziplinärer Modellierungsansatz*. Dissertation, Technische Universität München, 2005.

[GHH07] Alexander Gruler, Alexander Harhurin, and Judith Hartmann. Modeling the Functionality of Multi-Functional Software Systems . In *Proceedings of the 14th Annual IEEE International Conference on the Engineering of Computer Based Systems (ECBS)*, pages 349–358. IEEE Computer Society, 2007.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GK82] S. Goldsack and J. Kramer. Invariants in the Application-oriented Specification of Control Systems. *Automatica*, 18(2):71–76, 1982.

[GKRB96] Radu Grosu, Cornel Klein, Bernhard Rumpe, and Manfred Broy. State Transition Diagrams. Technical Report TUM-I9630, Technische Universität München, 1996.

[GM99] J. Paul Gibson and Dominique Mery. Formal Modelling of Services for Getting a Better Understanding of the Feature Interaction Problem. In *Ershov Memorial Conference*, pages 155–179, 1999.

[GM09] Alexander Gruler and Michael Meisinger. Hierarchical Decomposition of Multi-Functional Systems: Theoretical Fundamentals and Methodological Application. In *Proceedings of the 3rd International Conference on Fundamentals of Software Engineering (FSEN 2009)*, LNCS. Springer, 2009.

[GME06] Generic Modeling Environment. Website, 2006. `http://www.isis.vanderbilt.edu/projects/gme/`.

[GMP+03] Michael Gnatz, Frank Marschall, Gerhard Popp, Andreas Rausch, and Wolfgang Schwerin. The living software development process. *Software Quality Professional*, 5(3), June 2003.

[Gna05] Michael Andreas Josef Gnatz. *Vom Vorgehensmodell zum Projektplan*. Dissertation, Technische Universität München, 2005.

[Gri03] Klaus Grimm. Software technology in an automotive company: major challenges. In *Proceedings of the 25th international conference on Software engineering*, pages 498–503. IEEE Computer Society, 2003.

[Gro07] M.P. Groover. *Automation, production systems, and computer-integrated manufacturing*. Prentice Hall, 2007.

[Gru10] Alexander M Gruler. *A formal approach to software product families*. Dissertation, Technische Universität München, 2010.

[GS94] D. Garlan and M. Shaw. *An Introduction to Software Architecture*. School of Computer Science, Carnegie Mellon University, 1994.

[Gur00] Y. Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Transactions on Computational Logic (TOCL)*, 1(1):77–111, 2000.

[HB05] D. Herzberg and M. Broy. Modeling layered distributed communication systems. *Formal Aspects of Computing*, 17(1):1–18, 2005.

[HIM00] Jean Hartmann, Claudio Imoberdorf, and Michael Meisinger. UML-based Integration Testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2000.

[HKB09] Wolfgang Haberl, Stefan Kugele, and Uwe Baumgarten. Reliable Operating Modes for Distributed Embedded Systems. In *Model-Based Methodologies for Pervasive and Embedded Software, 2009. MOMPES'09. ICSE Workshop on*, pages 11–21. IEEE, 2009.

[HKN+07] C. Hofmeister, P. Kruchten, R.L. Nord, H. Obbink, A. Ran, and P. America. A general model of software architecture design derived from five industrial approaches. *The Journal of Systems & Software*, 80(1):106–126, 2007.

[HMU01] J.E. Hopcroft, R. Motwani, and J.D. Ullman. Introduction to automata theory, languages, and computation. *ACM SIGACT News*, 32(1):60–65, 2001.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[Hol03] G.J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison Wesley, 2003.

[HP98] David Harel and Michal Politi. *Modeling reactive systems with statecharts: the STATEMATE approach*. McGraw-Hill, Inc., 1998.

[HRSW06] Judith Hartmann, Sabine Rittmann, Peter Scholz, and Doris Wild. Formal incremental requirements specification of service-oriented automotive software systems. In *Proceedings of the the Second International Symposium on Service Oriented System Engineering (SOSE 2006)*, 2006.

[HS97] Franz Huber and Bernhard Schätz. Rapid Prototyping with AutoFocus. In A. Wolisz, I. Schieferdecker, and A. Rennoch, editors, *Formale Beschreibungstechniken für verteilte Systeme, GI/ITG Fachgespräch*, pages 343–352. GMD Verlag (St. Augustin), 1997.

[HT05] Tony Hey and Anne E. Trefethen. Cyberinfrastructure for e-Science. *Science*, 308(5723):817–821, 2005.

[Hum08] Bernhard Humm. Was ist eigentlich ein Service? *GI Softwaretechnik-Trends*, 28(4):8–11, 2008.

[HW03] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003.

[Ini08] Ocean Observatories Initiative. *Integrated Observatory Applications Architecture Document, Final Network Design. Version of Nov 2008*. Ocean Observatories Initiative Cyberinfrastructure, 2008.

[InS07] InServe. Project Website, Technische Universität München. Website, 2007. `http://www4.in.tum.de/proj/inserve`.

[ISO04] ISO/IEC. ISO/IEC 15504:2004 Process assessment. Parts 1–7. Int. Org. for Standarization, Standand, 2004. `http://www.iso.org`.

[ISO08a] ISO. ISO 9001: Quality management systems – Requirements. Int. Org. for Standarization, Standand, 2008. `http://www.iso.org`.

[ISO08b] ISO/IEC. ISO/IEC 12207:2008 Systems and software engineering – Software life cycle processes. Int. Org. for Standarization, Standand, 2008. `http://www.iso.org`.

[ISO08c] ISO/IEC. ISO/IEC 15288:2008 Systems and software engineering – System life cycle processes. Int. Org. for Standarization, Standand, 2008. `http://www.iso.org`.

[IT96] ITU-TS. Recommendation Z.120: Message Sequence Chart (MSC). Geneva, 1996.

[Jac92] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley Longman Publishing Co., Redwood City, CA, 1992.

[JBR99] Ivar Jacobson, Grady Booch, and Jim Rumbaugh. *The Unified Software Development Process*. Addison Wesley Longman Publishing Co., Boston, MA, 1999.

[Jon99] Christopher B. Jones. Compositionality, Inference and Concurrency. *Milennial Perspectives in Computer Science*, pages 175–186, 1999.

[JSO15] JSON. Javascript object notation (json). Website, 2015. `http://json.org/`.

[JZ98] Michael Jackson and Pamela Zave. Distributed Feature Composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, XXIV(10):831–847, October 1998.

[KGSB99] Ingolf Krüger, Radu Grosu, Peter Scholz, and Manfred Broy. From MSCs to Statecharts. In Franz J. Rammig, editor, *Distributed and Parallel Embedded Systems*, pages 61–71. Kluwer Academic Publishers, 1999.

[Kle98] Cornel Klein. *Anforderungsspezifikation durch Transitionssysteme und Szenarien*. PhD thesis, Technische Universität München, 1998.

[Kle06] Ekkart Kleinod. Modellbasierte Systementwicklung in der Automobilindustrie – Das MOSES Projekt. Technical report, Fraunhofer Institut Software und Systemtechnik ISST, 2006. ISST-Bericht 77/06, `http://www.isst.fraunhofer.de/deutsch/download/10095_moses_gesamt.pdf`.

[KLM06] Ingolf Krüger, Gunny Lee, and Michael Meisinger. Automating Software Architecture Exploration with M2Aspects. In *Proceedings of the ICSE 2006 Workshop on Scenarios and State Machines (SCESM)*, 2006.

[KM04a] Fabrice Kordon and Lemoine Michel, editors. *Formal Methods for Embedded Distributed Systems*. Springer, 2004.

[KM04b] Ingolf Heiko Krüger and Reena Mathew. Systematic Development and Exploration of Service-Oriented Software Architectures. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 177–187. IEEE, 2004.

[KMLS05] Ingolf H. Krüger, Reena Mathew, Stefan Leue, and Tarja Systä. Component Synthesis from Service Specifications. In *Scenarios: Models, Transformations and Tools*, volume 3466 of *LNCS*, pages 255–277. Springer Verlag, 2005.

[KMM05] Ingolf Krüger, Reena Mathew, and Michael Meisinger. From Scenarios to Aspects: Exploring Product Lines. In *Proceedings of the ICSE 2005 Workshop on Scenarios and State Machines (SCESM)*, 2005.

[KMM06] Ingolf Krüger, Reena Mathew, and Michael Meisinger. Efficient Exploration of Service-Oriented Architectures using Aspects. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, 2006.

[KMM07a] Ingolf Krüger, Michael Meisinger, and Massimiliano Menarini. Applying Service-Oriented Development to Complex Systems: BART case study. In *Proceedings of the Monterey Workshop 2005*, number 4322 in LNCS. Springer, 2007.

[KMM07b] Ingolf Krüger, Michael Meisinger, and Massimiliano Menarini. Runtime Verification of Interactions: From MCSs to Aspects. In *Post-Proceedings of the Seventh Workshop on Runtime Verification RV'07*, number 4839 in LNCS, pages 63–74. Springer, 2007.

[KMM08] Ingolf Krüger, Michael Meisinger, and Massimiliano Menarini. Interaction-based Runtime Verification for Systems of Systems Integration. *Journal of Logic and Computation*, 2008.

[KMMP06] Ingolf Krüger, Michael Meisinger, Massimiliano Menarini, and Stephen Pasco. Rapid Systems of Systems Integration - Combining an Architecture-Centric Approach with Enterprise Service Bus Infrastructure. In *Proceedings of the 2006 IEEE International Conference on Information Reuse and Integration (IRI)*, pages 51–56. IEEE, 2006.

[KNP04] Ingolf Krüger, Edward C. Nelson, and Venkatesh Prasad. Service-based Software Development for Automotive Applications. In *CONVERGENCE 2004*, 2004.

[Kru00a] Philippe Kruchten. *The Rational Unified Process. An Introduction.* Addison-Wesley, 2nd edition, 2000.

[Krü00b] Ingolf Krüger. *Distributed System Design with Message Sequence Charts.* PhD thesis, Technische Universität München, 2000.

[Krü02a] Ingolf Krüger. Towards Precise Service Specification with UML and UML-RT. In *Critical Systems Development with UML (CSDUML).* Workshop at UML 2002, 2002.

[Krü02b] Ingolf H. Krüger. Specifying Services with UML and UML-RT. Foundations, Challenges and Limitations. *Electronic Notes in Theoretical Computer Science*, 65(7), 2002.

[Krü03] Ingolf H. Krüger. Capturing Overlapping, Triggered, and Preemptive Collaborations Using MSCs. In Mauro Pezzè, editor, *FASE 2003*, volume 2621 of *LNCS*, pages 387–402. Springer Verlag, 2003.

[Krü04] Ingolf Heiko Krüger. Service Specification with MSCs and Roles. In *Proceedings of the IASTED International Conference on Software Engineering*, 2004.

[KSLB03] Gabor Karsai, Janos Sztipanovits, Akos Ledeczi, and Ted Bapty. Model-Integrated Development of Embedded Software. In *Proceedings of IEEE January 2003*, 2003.

[KSTW04] Leonid Kof, Bernhard Schätz, Ingomar Thaler, and Alexander Wisspeintner. Service-based development of embedded systems. In *Net.Object Days Conference, OOSE Workshop*, Erfurt, 2004.

[KTMF09] Katarzyna Keahey, Mauricio Tsugawa, Andrea Matsunaga, and Jose Fortes. Sky Computing. *Internet Computing, IEEE*, 13(5):43–51, 2009.

[Küh05] Thomas Kühne. What is a model? In Jean Bézivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, volume 04101 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[KWGS03] Sharath Kodase, Shige Wang, Zonghua Gu, and Kang G. Shin. Improving Scalability of Task Allocation and Scheduling in Large Distributed Real-time Systems using Shared Buffers. In *Proceedings of the 9th IEEE Real-time/Embedded Technology and Applications Symposium (RTAS)*, 2003.

[Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 27(7):558–565, July 1978.

[Lam94] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lan14] Markus Lanthaler. *Third Generation Web API–Bridging the Gap between REST and Linked Data*. PhD thesis, Graz University of Technology, Austria. Institute of Information Systems and Computer Media, 2014.

[LCA02] K. Lano, D. Clark, and K. Androutsopoulos. RSDS, a subset of UML with precise semantics. *The fourth workshop on rigorous object-oriented methods, Kings College London, UK*, 2002.

[LDR06] O. Labbani, J.L. Dekeyser, and É. Rutten. Separating Control and Data Flow: Methodology and Automotive System Case Study. *INRIA*, 2006.

[Lev00] Nancy G. Leveson. System safety in computer-controlled automotive systems. In *SAE Congress*, March 2000.

[Ley01] Frank Leymann. Web Services Flow Language (WSFL 1.0). Technical report, IBM Corporation, 2001.

[LG12] Markus Lanthaler and Christian Gütl. On using json-ld to create evolvable restful services. In *Proceedings of the Third International Workshop on RESTful Design*, pages 25–32. ACM, 2012.

[LJS08] Daniel Lucrédio, Ethan K Jackson, and Wolfram Schulte. Playing with Fire: Harnessing the Hottest Technologies for Ultra-Large-Scale Systems. In *15th Monterey Workshop-Foundations of Computer Software, Future Trends and Techniques for Development*, 2008.

[LKA+95] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4), 1995.

[Mat04] Reena Mathew. Systematic Definition, Implementation and Evaluation of Service-Oriented Software. Master's thesis, University of California, San Diego, 2004.

[MCB+11] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, Angela Hung Byers, and McKinsey Global Institute. *Big data: The next frontier for innovation, competition, and productivity*. McKinsey Global Institute San Francisco, 2011.

[Men12] Massimiliano Menarini. *Composing Crosscutting Concerns: A Service-Oriented View*. Ph.d. dissertation, University of California, San Diego, 2012.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[MK07] Michael Meisinger and Ingolf Krüger. A Service-Oriented Extension of the V-Modell XT. In *Proceedings of the 14th Annual IEEE International Conference on the Engineering of Computer Based Systems (ECBS'07)*, Tucson AZ, 2007. IEEE.

[MKF10] Paul Marshall, Kate Keahey, and Tim Freeman. Elastic site: Using clouds to elastically extend site resources. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 43–52. IEEE Computer Society, 2010.

[Möl99] Bernhard Möller. Algebraic Structures for Program Calculation. In Manfred Broy and Ralf Steinbrüggen, editors, *Calculational System Design*, volume 173 of *NATO Science Series F*, pages 25–97. IOS Press, 1999.

[Moo56] E.F. Moore. Gedanken-experiments on sequential machines. *Automata Studies*, 34:129–153, 1956.

[MR03] F. Maraninchi and Y. Rémond. Mode-Automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, 46(3):219–254, 2003.

[MR08] Michael Meisinger and Sabine Rittmann. A comparison of service-oriented development approaches. Technical Report TUM-I0825, Technische Universität München, 2008.

[MRD+04] Michael Meisinger, Andreas Rausch, Martin Deubler, Michael Gnatz, Ulrike Hammerschall, Inga Küffer, and Sascha Vogel. Das V Modell 200x - ein modulares Vorgehensmodell. In Ralf Kneuper, Roland Petrasch, and Manuela Wiemers, editors, *11. Workshop der Fachgruppe WI-VM der Gesellschaft für Informatik e.V. (GI) zur Akzeptanz von Vorgehensmodellen*. Shaker Verlag, 2004.

[MRS06] M. Meisinger, A. Rausch, and M. Sihling. 4everedit – Team-based Process Documentation Management. *Software Process Improvement and Practice*, 11(6):627–642, 2006.

[MS] Tiziana Margaria and Bernhard Steffen. Service Engineering: Linking Business and IT. *Computer*, 39(10).

[MSZ01] S.A. McIlraith, T.C. Son, and H. Zeng. Semantic Web Services. *IEEE INTELLIGENT SYSTEMS*, pages 46–53, 2001.

[New02] Eric Newcomer. *Understanding Web services: XML, WSDL, SOAP, and UDDI*. Addison-Wesley, May 2002.

[New15] Sam Newman. *Building Microservices*. "O'Reilly Media, Inc.", 2015.

[Nor00] O.S. Noran. Mapping of ISO 15288 and ISO 12207 to ISO 15704. Master's thesis, Griffith University, School of Computing and Information Technology, 2000.

[NP03] Edward C Nelson and K V Prasaad. Automotive Infotronics: An emerging domain for Service-Based Architecture. In I. H. Krüger, B. Schätz, M. Broy, and H. Hussmann, editors, *SBSE'03 Service-Based Software Engineering, Proceedings of the FM2003 Workshop*, Technical Report TUM-I0315, pages 3–14. Technische Universität München, 2003.

[NSKB03] Sandeep Neema, Janos Sztipanovits, Gabor Karsai, and Ken Butts. Constraint-Based Design-Space Exploration and Model Synthesis. In *EMSOFT 2003*, pages 290–305, 2003.

[OAS05a] OASIS. ebXML. Website, 25-Nov 2005. `http://www.ebxml.org/`.

[OAS05b] OASIS. UDDI Specification, 25-Nov 2005. `http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm`.

[OAS06] OASIS. Web Services Business Process Execution Language (WS-BPEL), Version 2.0. Specification public draft, 23-Aug-2006, 2006. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.pdf`.

[OHE97] Robert Orfali, Dan Harkey, and Jeri Edwards. *Instant CORBA*. John Wiley & Sons, 1997.

[OMG] OMG (Object Management Group). Software process engineering meta-model, version 2.0.

[OMG03a] OMG (Object Management Group). Model Driven Architecture (MDA). MDA Guide 1.0.1, omg/03-06-01, 2003. `http://www.omg.org/mda`.

[OMG03b] OMG (Object Management Group). UML Profile for Schedulability, Performance, and Time. ptc/2003-03-02 OMG Specification, 2003.

[OMG06] OMG (Object Management Group). Meta Object Facility (MOF), version 2.0. formal/2006-01-01 OMG Specification, 2006.

[OMG11a] OMG (Object Management Group). MOF 2 XMI Mapping (XMI), Version 2.4.1. OMG Specification formal/2011-08-09 (MOF/XMI specification), 2011. `http://www.omg.org/spec/XMI/`.

[OMG11b] OMG (Object Management Group). UML, Version 2.4.1. OMG Specification formal/2011-08-06 (superstructure) and formal/2011-08-05 (infrastructure), 2011.

[OOI14] OOI Cyberinfrastructure. OOINet Architecture Specification. Website, 2014. `https://confluence.oceanobservatories.org/display/syseng/Architecture+and+Design`.

[Ora13] Oracle Corporation. Java GlassFish. Website, 2013. `http://glassfish.java.net/`.

[Ost87] L. Osterweil. Software processes are software too. In *Proceedings of the 9th international conference on Software Engineering*, pages 2–13. IEEE Computer Society Press Los Alamitos, CA, USA, 1987.

[OWL04a] W3C: OWL: Web Ontology Language Overview, W3C Recommendation 10-Feb-2004, 2004. `http://www.w3.org/TR/owl-features/`.

[OWL04b] W3C: OWL-S: Semantic Markup for Web Services, W3C Member Submission 22 November 2004, 2004. `http://www.w3.org/Submission/OWL-S/`.

[Par79] David L. Parnas. *On the criteria to be used in decomposing systems into modules*. Yourdon Press Upper Saddle River, NJ, USA, 1979.

[Pau09] Cesare Pautasso. RESTful Web service composition with BPEL for REST. *Data & Knowledge Engineering*, 68(9):851–866, 2009.

[PB01] David S. Platt and Keith Ballinger. *Introducing Microsoft .NET*. Microsoft Press, 2001.

[Pel01] Doron A. Peled. *Software Reliability Methods*. Springer, 2001.

[Pen11] Birgit Penzenstadler. *DeSyRe: Decomposition of Systems and their Requirements*. Dissertation, Technische Universität München, 2011.

[PR02] Klaus Pohl and Andreas Reuys. Considering Variabilities during Component Selection in Product Family Development. In *Proceedings of the 4th International Workshop on Product Family Engineering (PFE-4)*, volume 2290 of *LNCS*, pages 22–37. Springer Verlag, 2002.

[PR03] Jan Philipps and Bernhard Rumpe. Refactoring of programs and specifications. In *Practical Foundations of Business System Specifications*, pages 281–297. Springer, 2003.

[Rau02] Andreas Rausch. "Design by Contract" + "Componentware" = "Design by Signed Contract". *Journal of Object Technology*, 1(3):19–36, 2002.

[RB06] Andreas Rausch and Manfred Broy. *Das V-Modell XT. Grundlagen, Erfahrungen und Werkzeuge.* dpunkt.verlag, 2006. In German.

[RBTK05] Andreas Rausch, Christian Bartelt, Thomas Ternite, and Marco Kuhrmann. The V-Modell XT Applied - Model-Driven and Document-Centric Development. In *Proceedings of the 3rd World Congress for Software Quality, Volume III, Online Supplement*, 2005.

[RC95] Mary Beth Rosson and John M. Carroll. Narrowing the Specification-Implementation Gap in Scenario-Based Design. In *Scenario-based design: envisioning work and technology in system development*, pages 247–278, New York, NY, USA, 1995. John Wiley & Sons, Inc.

[RFH+05] Sabine Rittmann, Andreas Fleischmann, Judith Hartmann, Christan Pfaller, Martin Rappl, and Doris Wild. Integrating Service Specifications on Different Levels of Abstraction. In *IEEE International Workshop on Service-Oriented System Engineering (SOSE)*. IEEE, IEEE, 2005.

[Rit04] Sabine Rittmann. Exploring Service-Oriented Software Development for Automotive Systems. Master's thesis, Technische Universität München, 2004.

[Rit08] Sabine Rittmann. *A methodology for modeling usage behavior of multifunctional systems.* Dissertation, Technische Universität München, 2008.

[RKL+05] Dumitru Roman, Uwe Keller, Holger Lausen, Ruben Lara Jos de Bruijn, Michael Stollberg, Axel Polleres, Cristina Feier, Christoph Bussler, and Dieter Fensel. Web service modeling ontology. *Applied Ontology*, 1(1):77–106, 2005.

[Roy70] Winston W Royce. Managing the development of large software systems. *Proceedings of IEEE Wescon*, 26(8), 1970.

[RRMP08] A. Rausch, R. Reussner, R. Mirandola, and F. Plasil, editors. *The Common Component Modeling Example. Comparing Software Component Models*, volume 5153 of *LNCS*. Springer, 2008.

[SAB+13] Munindar P Singh, Matthew Arrott, Tina Balke, Amit Chopra, RMJ Christiaanse, Stephen Cranefield, Frank Dignum, Davide Eynard, Emilia Farcas, Michael Meisinger, et al. The uses of norms. In *Dagstuhl Follow-Ups, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Volume 4*. Dagstuhl, 2013.

[Sal02] Christian Salzmann. *Modellbasierter Entwurf spontaner Komponentensysteme*. PhD thesis, Technische Universität München, 2002.

[SBK+05] Douglas C. Schmidt, Krishnakumar Balasubramanian, Arvind S. Krishna, Emre Turkay, and Aniruddha Gokhale. Model Driven Development for Distributed Real-time and Embedded Systems. In Sebastien Gerard, Joel Champea, and Jean-Philippe Babau, editors, *Model Driven Development for Distributed Real-time and Embedded Systems*. Hermes, 2005.

[Sch98] Peter Scholz. *Design of Reactive Systems and their Distributed Implementation with Statecharts*. PhD thesis, Technische Universität München, 1998.

[Sch02] Bernhard Schätz. Towards Service-Based Systems Engineering: Formalizing and mu-Checking Service Specifications. Technical Report TUMI-0206, Technische Universität München, 2002.

[Sch04] Klaus Schneider. *Verification of Reactive Systems: Formal Methods and Algorithms*. Springer, 2004.

[Sch06a] Bernhard Schätz. Combining Product Lines and Model-Based Development. In *Proceedings of Formal Aspects of Component Systems (FACS 2006)*. Electronic Notes in Theoretical Computer Science, 2006.

[Sch06b] Bernhard Schätz. Model-Based Engineering of Embedded Control Software. In *Proceedings of MDB/MOMPES 2006*. IEEE Computer Society, 2006.

[Sch06c] Douglas C. Schmidt. Model-Driven Engineering. *Computer*, 39(2):25–31, 2006.

[Sch08] Bernhard Schätz. Modular functional descriptions. *Electronic Notes in Theoretical Computer Science*, 215:23–38, 2008.

[Sci15] SciON Contributors. Scientific Observatory Network (SciON). Website, 2015. `https://github.com/scionrep/scioncc`.

[SD05] Zoran Stojanovic and Ajantha Dahanayake. *Service-Oriented Software System Engineering: Challenges and Practices*. IGI Publishing, Hershey, PA, USA, 2005.

[Sel03] Bran Selic. The pragmatics of model-driven development. *Software, IEEE*, 20(5):19–25, 2003.

[Ses97] Roger Sessions. *COM and DCOM*. John Wiley & Sons, 1997.

[SFGP05] Bernhard Schätz, Andreas Fleischmann, Eva Geisberger, and Markus Pister. Model-Based Requirements Engineering with AutoRAID. In Armin B. Cremers, Rainer Manthey, Peter Martini, and Volker Steinhage, editors, *Workshop "Modellbasierte Qualitätssicherung" (QUAM)*, Lecture Notes in Informatics (LNI), pages 511–516. Bonner Köllen Verlag, 2005.

[SG96] M. Shaw and D. Garlan. *Software Architecture, Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[SGO+10] O. Schofield, S. Glenn, J. Orcutt, M. Arrott, M. Meisinger, A. Gangopadhyay, W. Brown, R. Signell, M. Moline, Y. Chao, et al. Automated Sensor Networks to Advance Ocean Science. *EOS Transactions*, 91:345–346, 2010.

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. Wiley, 1994.

[SH05] M.P. Singh and M.N. Huhns. *Service-Oriented Computing: Semantics, Processes, Agents*. John Wiley and Sons, 2005.

[Sha02] Robert Shapiro. A Comparison of XPDL, BPML, and BPEL4WS, 2002. `http://xml.coverpages.org/Shapiro-XPDL.pdf`.

[SK98] I. Sommerville and G. Kotonya. *Requirements Engineering: Processes and Techniques*. John Wiley & Sons, Inc., 1998.

[Sof07] Software product lines. Software Engineering Institute, 2007. `http://www.sei.cmu.edu/productlines/index.html`.

[Som89] Ian Sommerville. *Software Engineering*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1989.

[SPHP02] Bernhard Schätz, Alexander Pretschner, Franz Huber, and Jan Philipps. Model-based Development of Embedded Systems. Technical Report TUMI-0402, Technische Universität München, 2002.

[SS03] C. Salzmann and B. Schätz. Service based Software Specification. In *Proceedings of the Int. Workshop on Test and Analysis of Component Based Systems (TACOS), ETAPS 2003*. IEEE Computer Society, 2003.

[Sta73] H. Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag, 1973. in German.

[Ste97] Robert Stephens. A Survey of Stream Processing. *Acta Informatica*, 34:491–541, 1997.

[STK02] James Snell, Doug Tidwell, and Pavel Kulchenko. *Programming Web Services with SOAP*. O'Reilly, 2002.

[Stö01] Harald Störrle. Describing Process Patterns with UML. In *Proc. of the 8th European Workshop on Software Process Technology (EWSPT '01)*, pages 173–182. Springer, 2001.

[SUN06] SUN Microsystems Inc. Java Platform, Enterprise Edition (Java EE, J2EE). Website, 15-Nov 2006. `http://java.sun.com/javaee/`.

[Sys06] Systems Modeling Language (SysML), 2006. `http://www.sysml.org/`.

[SZP+03] John A. Stankovic, Ruiqing Zhu, Ramasubramaniam Poornalingam, Chenyang Lu, Zhendong Yu, Marty Humphrey, and Brian Ellis. VEST: An Aspect-based Composition Tool for Real-time Systems. In *Proceedings of the IEEE Real-time Applications Symposium*, 2003.

[Szy02] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

[TMA+96] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A Component and Messages Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, 22(6):390–406, June 1996.

[TRH+04] David Trowbridge, Ulrich Roxburgh, Gregor Hohpe, Dragos Manolescu, and E.G. Nadhan. *Integration Patterns. Patterns & Practices*. Microsoft Press, 2004.

[Tur98] C. Turner. Feature Engineering of Software Systems, 1998.

[TVS01] A.S. Tanenbaum and M. Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2001.

[vBRS03] M. von der Beeck, P. Braun, M. Rappl, and C. Schröder. Automotive UML: a (meta) model-based approach for systems development. In *UML for real: design of embedded real-time systems*, pages 271–299. Kluwer Academic Publishers, 2003.

[vdA03] Wil M.P. van der Aalst. Don't go with the flow: Web services composition standards exposed. *IEEE Intelligent Systems*, 18(1):72–76, 2003.

[vL04] Axel van Lamsweerde. Goal-oriented requirements enginering: a roundtrip from research to practice. *Requirements Engineering Conference, 2004. Proceedings. 12th IEEE International*, pages 4–7, 2004.

[Vog03] Werner Vogels. Web Services Are Not Distributed Objects. *IEEE Internet Computing*, 7(6):59–66, 2003.

[Vog15] Andreas Vogelsang. *Model-based Requirements Engineering for Multifunctional Systems*. Dissertation, Technische Universität München, 2015.

[W3C01] W3C. Web Services Description Language (WSDL), Version 1.1. W3C Note, 15-Mar-2001, 2001. `http://www.w3.org/TR/wsdl/`.

[W3C02] W3C. Web Service Choreography Interface (WSCI), Version 1.0. W3C Note, 8-Aug-2002, 2002. `http://www.w3.org/TR/wsci/`.

[W3C04a] W3C. W3C: Resource Description Framework (RDF). Primer, Concepts, Syntax, Semantics, Vocabulary, Test Cases, 11-Feb 2004. `http://www.w3.org/rdf`.

[W3C04b] W3C. Web Services Architecture, 11-Feb 2004. `http://www.w3.org/TR/ws-arch/`.

[W3C05] W3C: Web Service Semantics - WSDL-S, Version 1.0. W3C Member Submission, 7-Nov-2005, 2005. `http://www.w3.org/Submission/WSDL-S/`.

[Wal02] Aaron Walsh. *UDDI, SOAP, and WSDL: The Web Services Specification Reference Book*. Prentice Hall, 2002.

[WB09] David Waltz and Bruce G. Buchanan. Automating Science. *Science*, 324(5923):43–44, 2009.

[WFH+06] Doris Wild, Andreas Fleischmann, Judith Hartmann, Christian Pfaller, Martin Rappl, and Sabine Rittmann. An Architecture-Centric Approach towards the Construction of Dependable Automotive Software. In *Proceedings of the SAE 2006 World Congress*, 2006.

[Wie99] Karl E. Wiegers. *Software Requirements*. Microsoft Press Redmond, 1999.

[WK04] Jon Whittle and Ingolf Krüger. A Methodology for Scenario-Based Requirements Capture. In *Proceedings of the 4th Int. Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (SCESM 2004)*, 2004.

[WKM04] Victor Winter, Fabrice Kordon, and Lemoine Michel. The BART Case Study. In Fabrice Kordon and Lemoine Michel, editors, *Formal Methods for Embedded Distributed Systems*, pages 3–22. Springer, 2004.

[WM06] Stefan Wagner and Michael Meisinger. Integrating a Model of Analytical Quality Assurance into the V-Modell XT. In *Proceedings of the 3rd International Workshop on Software Quality Assurance (SoQUA 2006)*. IEEE, IEEE, 2006.

[Zac06] John Zachman. *The zachman framework for enterprise architecture*. Zachman Framework Associates, 2006.

[Zav93] Pamela Zave. Feature Interactions and Formal Specifications in Telecommunications. *Computer*, 26(8):20–29, 1993.

[Zav01] Pamela Zave. Feature-Oriented Description, Formal Methods, and DFC. In *Proceedings of the FIREworks Workshop on Language Constructs for Describing Features*, pages 11–26. Springer-Verlag, 2001.

[ZJ97] Pamela Zave and Michael Jackson. Telecommunications Service Requirements: Principles for Managing Complexity. *Requir. Eng.*, 2(2):92–101, 1997.

[ZJ01] Pamela Zave and Michael Jackson. New feature interactions in mobile and multimedia telecommunication services. In *Feature Interactions in Telecommunications and Software Systems VI*, pages 51–66. IOS Press, 2001. `http://www.research.att.com/~pamela/fiw6.pdf`.

[ZJ03] Pamela Zave and Michael Jackson. The DFC Manual. Technical report, AT&T, November 2003. `http://www.research.att.com/~pamela/man.pdf`.

[ZKG04] Olaf Zimmermann, Pal Krogdahl, and Clive Gee. Elements of Service-Oriented Analysis and Design. Technical report, IBM developerWorks, 2004. Version of 2-Jun-2004, `http://www-128.ibm.com/developerworks/webservices/library/ws-soad1/`.