T U M

I N S T I T U T F Ü R I N F O R M A T I K

A comparison of service-oriented development approaches

Michael Meisinger, Sabine Rittmann

TUM-I0825
August 08

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-08-I0825-0/0.-FI
Alle Rechte vorbehalten
Nachdruck auch auszugsweise verboten

©2008

Druck: Institut für Informatik der
Technischen Universität München

Abstract. Software enabled systems are essential in many applications domains, such as business information systems, complex control software and telecommunication systems; they are quickly emerging in others, for instance in embedded systems and mobile network applications. Such systems are ever growing in size and complexity, with simultaneously increasing demands on quality, performance, governance and correctness. The number of complex systems of systems integration and maintenance efforts is rising. It is of paramount importance to manage the development of such systems in an effective, timely, and resource saving way. It is mandatory to reduce complexity by raising the levels of abstraction as possible and to have effective system development models that support it.

Service-orientation is a promising idea of managing the development, realization and implementation of complex software enabled systems. General idea is to focus on the functionality of a system delivered by interactions with its users – its services – and to abstract from more concrete details, such as structural and implementation aspects. Many development approaches in academia and industry label themselves as service-oriented. Application of services ranges from early requirements engineering phases to the deployment of executable systems. However, there is no common understanding of the meaning of the terms service and service-orientation, even within one application domain.

In this article, we provide an overview of the underlying concepts of service-orientation, depict potential benefits across development phases and survey existing service-oriented development approaches. We identify criteria that enable us to compare the approaches and show their commonalities and differences. We conclude the essence of service-orientated and related approaches and their benefits.

Keywords: Services, service-orientation, service-oriented architectures, model-based development, survey

Table of Contents

1. Introduction 2
  1.1. Contribution 3
  1.2. Outline 4
  1.3. Acknowledgements 4
2. Services throughout the Development Process 4
  2.1. Services in Requirements Engineering 5
  2.2. Services in Modeling and Architecture Design 6
  2.3. Services in Implementation and Integration 7
  2.4. Services in deployment and run-time 8
3. Service-Orientation in Specific Application Domains 10
  3.1. Business Information Systems and Web Services 10
    3.1.1. Basic Concepts 11
    3.1.2. Definition of Web Services 13
  3.2. Telecommunication Systems 14
4. Service-Oriented Approaches at TUM 16
  4.1. The InServe Methodology 16
  4.2. MEwaDis 19
  4.3. Services in mobilSoft 19
  4.4. Functional Architecture Modeling 21
  4.5. The FOCUS/JANUS Approach 23
5. Comparison and Evaluation 26
  5.1. Evaluation Criteria 26
  5.2. Comparison Table 33
6. Discussion 35
7. Summary and Outlook 35
8. Bibliography 36

1. Introduction

The complexity of software systems and of their development is continuously increasing, because of growing system size, the demands for higher quality, an increasing degree of distribution, the increasing heterogeneity of software architectures just to name a few. In particular, reactive systems become more and more distributed across networks, and spread across multiple different processes, nodes, control units and components. The increasing distribution leads to a higher degree of concurrency within such systems and significantly higher interaction complexity. The always growing demand for interoperation among software systems, especially across operational boundaries, also results in challenging problems. The heterogeneity of distributed systems and their sophisticated interactions must be managed systematically to keep development and

maintenance effective. A promising approach to address these challenges is the *service-oriented paradigm,* because it provides means to structure system functionality into manageable self-contained pieces – the services with interactions as the uniform concepts to deliver services; furthermore it separates functionality from technical details such as component distribution over networks. Services and their dependencies can be specified on a logical level and later in the development process be mapped to actual deployment infrastructures.

The term *service* is used often and extensively within the community – in research and in industry. Many service-oriented technologies, infrastructures and approaches exist, which often target different problems or focus on different application domains. Common to most approaches is the lack of a sufficiently precise definition of the term service. In the literature, there are often similar ideas, which have been developed into service-oriented approaches with related concepts. Among these, for instance, are web services [1] [2] and service-oriented architectures, feature-oriented systems in the telecommunication area [3][4], and services in the context of middleware technologies [5][6]. Often, confusion is the result when comparing different approaches and methods. Questions such as the following arise:

- Which domain is an approach considered for?
- What is the special focus of a service-based technique?
- What are the main concepts of an approach, besides services?
- What are the advantages and disadvantages of the approach, respectively?
- What part of the development process is addressed by an approach?
- Is it a research result or an approach that is targeted for direct application?

At the Software and Systems Engineering Lab of the Technische Universität München, we have done extensive research and application projects in the area of service-orientation [7], software engineering and embedded systems design. We are involved in several projects dealing with service-orientation (see [8], [9], [10] for a selection). Even within these projects, we observe a variety of different notions of service and service-oriented methodologies.

1.1. *Contribution*

In this article we provide an *overview* of existing service-oriented approaches and concepts, and a *comparison* and *evaluation* of them. We do not aim for completeness; instead, we strive to give a good overview of current service notions from the literature and our group. This overview should help to understand the problems of the different approaches which are often strongly influenced by the constraints of the underlying domain. This can lead to an improved understanding of the focus of the respective work.

We achieve this by identifying criteria based upon which we evaluate the service-oriented approaches. The result is a structured overview of the commonalities and differences of the approaches. We isolate the essence of service-orientation and conclude benefits of the service paradigm.

1.2. *Outline*

The article is structured as follows. In Section 2, we present an overview of the usefulness of service-oriented approaches in the different stages of the development cycle, at various levels of abstraction. In Section 3, we highlight certain application domains in which notions of services and similar concepts are successful or emerging. Section 4 presents different service-oriented development approaches used within the Software & Systems Engineering Lab of the Technische Universität München. In Section 5, we introduce a list of 15 evaluation criteria, which we use to compare the previously described notions of service. We present the comparison in form of a table. In Section 6, we provide a conclusion based on the comparison and Section 7 summarizes our efforts and presents an outlook into future work.

1.3. *Acknowledgements*

This work was partially funded by the Deutsche Forschungsgemeinschaft (DFG) as part of the project “InServe” and by the High-tech Offensive Bayern in the project “mobilSoft”.

2. Services throughout the Development Process

The term *service* is used in many different contexts in software and systems engineering and may address quite different concepts. Talking about service-orientation, service-based development and service-oriented architectures does not always guarantee a common understanding. Often, the meaning depends on the application domain, such as business information systems where the notion of web service is prevalent, or mobile systems, where services are often seen as context-aware and adaptive functionality. Another important delimiter is the level of abstraction that commonly goes along with the progression in the development process.

Figure 1 shows the development process divided into four main phases: requirements analysis and specification, architecture and design, implementation and integration, deployment and execution. Each of these phases can be associated with certain levels of abstraction and certain sets of concepts needed and applied. The figure shows a non-exhaustive list of concepts that are of concern in the respective phases.

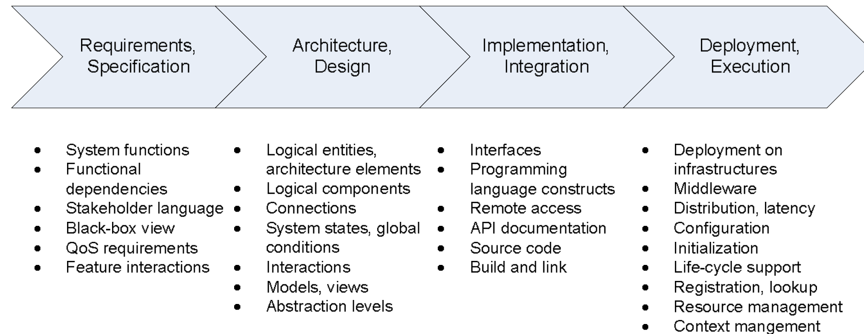

Figure 1: Development phases and primary concerns

In the following, we briefly characterize each of the four development phases, describe the roles that service-orientation might play in these phases and the benefits that such approaches might bring (see also [7]).

2.1. *Services in Requirements Engineering*

Requirements Engineering. Requirements engineering is concerned with capturing and agreeing on the functional and non-functional properties of systems in a structured way. Classical ways to conduct requirements elicitation and analysis is via structured natural language text (using text-based tools such as Word, Doors, etc.). Usage scenarios can be captured as use cases and dependencies depicted as use case diagrams.

The role of services in requirements engineering. A multi-functional system can be understood as a system offering a number of separate, partially mutually dependent services (functions). This idea is related to capturing the different use cases of a system. Services are pieces of functionality that each provide a partial view on the functionality of the system under certain aspects of usage. Their entirety, considering all service dependencies and interactions, makes up the system functionality. Non-functional requirements can be associated with the system or its services. Synonyms for this interpretation of service are system function, feature, use case, scenario etc. Services can vary in their granularity or degree of preciseness. In case of a precisely defined notion of service, the functional properties of the system can be captured in terms of services. Services can depend on other services. The type of dependency can be further distinguished. Services might, for instance, “interact” or “control” or “influence” each other. These dependencies can be depicted in a service dependency graph [11][12].

Benefits. Possible applications of such structured, formalized service models are consistency and completeness analysis, predictions of certain properties for the realized system (e.g. by means of a simulation of formal service models). Certain forms of feature interaction can be detected and considered by analyzing a service dependency graph.

Additionally, a precisely elaborated service repository can provide the basis for a thorough, formal system specification and can provide a precise foundation for requirements tracing spanning the design model and implementation.

2.2. *Services in Modeling and Architecture Design*

Modeling and architecture design. Model-based development [13][14][15] helps to handle the complexity that comes with the development of distributed software systems. Precisely defined models provide abstractions and notational elements for all stakeholders throughout the development cycle. Models are supposed to provide multiple consistent views on the system and its architecture on different levels of abstraction, tailored for specific intents. The architecture of a reactive system is an essential design artifact in the development process; it needs to effectively support all services of the system in often heterogeneous, distributed environments. The decomposition of a system into its parts and their interconnections determines the further quality of the system, influencing properties such as performance, robustness, maintainability, flexibility to change etc.

The role of services within the context of modeling and architecture design. Services can be used as a way of structuring both the modeling process and the models. In service-based approaches to architecture, services are the design entities that drive architecture development and component identification. Services capture defined pieces of functionality and are associated with elements of the software architecture (components, patterns). In fact, services model functionality provided by an interplay of collaborating architectural entities. A promising way to describe interactions in the course of service delivery between independent entities are sequence diagrams or MSCs [16].

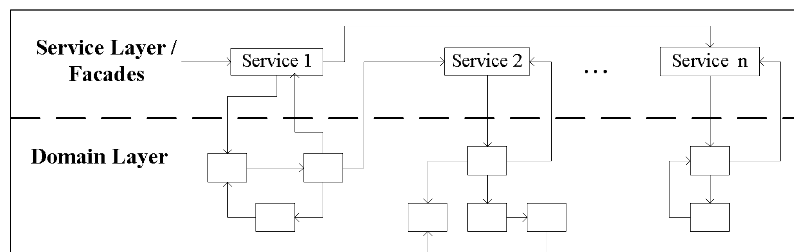

Figure 2: Service-Oriented Architectures

The center of concern in model-based design has so far been individual components rather than their interplay. A service-oriented development approach can be seen as a seamless extension of component-based development towards a higher level of abstraction and functional view on the system with system-wide, component crosscutting functionality. The concept of service decouples abstract behavior from the implementation architectures, by emphasizing the interactions among components.

Figure 2 shows a typical “layout” of an application composed as a set of services. Often such systems consist of at least two distinct layers: one domain layer, which houses all domain objects and their associated logic; and one service layer, which acts as a façade to the underlying domain objects – in effect, offering an interface that shields the domain objects from client software. Typically, services coordinate workflows among the domain objects; they call, and thus depend on, other services. Some of the services, say Service 1 and Service 2 in our example, reside on the same physical machine, whereas others, such as Service n are accessible remotely via the Internet. In later steps of system design, services are mapped freely to existing or newly designed component configurations and architectures. Components can provide an individual or multiple services as long as the service interfaces are consistent.

Benefits. Designing and evaluating architectures of a system is a critical part in system development. The architecture determines the basic non-functional properties of such a system, such as scalability, reliability, security, robustness, maintainability etc. Certain architectures turn out as more effective to support the specified services of the system than others. Designing the architectures with a service-oriented paradigm in mind allows designers to abstract from technical interplays of components and concentrate on the services that are provided to the environment. Such architectures promise to be, in general, more flexible and robust to changing environments and requirements. Systems are composed as an interplay or orchestration of different services.

Usually, stakeholders express their needs in terms of functionality. E.g. they speak of features or functionality to be changed, added, etc. However, the traditional component-based development is concerned about structure, rather than functionality. The consequence is a conceptual gap between requirements engineering and design. The introduction of service-orientation before the actual component-based design takes place promises to make the transition between RE and design (more) seamless and less cumbersome.

2.3. *Services in Implementation and Integration*

Implementation. In the implementation phase of the development process, the design model is realized by actual hardware and program code. Implementation varies significantly depending on the applied strategy. In case of model-based development with comprehensive and precise design models, code generation can be performed (semi-) automatically. Manual coding where applied needs to follow the design model to fulfill the designed properties. Implementing a distributed system based on a design model and component architecture is a difficult task. Especially the integration of components requires much care and effort to create consistent, efficient and homogenous systems. Important concepts here are packaging and robustness of components, availability of suitable syntactic and semantic component interfaces, powerful tools and flexible infrastructures as the basis of implementation etc.

Services in the context of implementation. Existing service technologies enable to define services as functional interfaces of components or objects. Functionality is encapsulated behind such service interfaces on a suitable level of abstraction and granularity. Such service implementations are usually stateless; they do not depend on the caller or environment state besides the parameters given during the service invocation. This provides a high degree of deployment flexibility, robustness and decoupling. Service-oriented middleware can provide local or remote access to the services and applications can be structured as an interplay or workflow of multiple service invocations.

Web service technology, for instance, defines standards, protocols and methodology to provide well defined functionalities as web services, to describe the interfaces using WSDL [17], and to connect services using a service-oriented middleware, such as DotNET [1] to form service-oriented distributed applications.

Feature-driven development makes features the center of concern. Systems are specified and developed in terms of independent features that are later integrated to form a complete system. The features are based on a system core, providing the basic functionality. Features depend on and overlay each other. There needs to be a defined process to define the dependencies and interactions of the features in order to maintain the consistency and coherence of the system. In such systems, as they occur in the telecommunications domain, extensions happen by adding new features that modify existing feature functionality or that add new functionality to the system. Features or services are thus the basic units of increment and change in the system development process and implementation.

Benefits. Providing services as high-level, stateless encapsulations of functionality and providing remote access points within distributed systems is an important means to decouple distributed system entities. This raises the level of interaction to a level that allows for easy interoperability in heterogeneous system environments. Pieces of the system are decoupled from the rest and can be replaced and modified as long as they are adhering to the service protocol. High level, widely accepted standard, such as XML and SOAP help to decrease the burden of systems integration. Message-oriented architectures based on service invocations further decouple systems by supporting asynchrony and facilitate system integration.

2.4. *Services in deployment and run-time*

Deployment and run-time. In order for a system to be executed, the executable code – mostly packaged in components – needs to be distributed across computational nodes, processes, electronic control units, etc. The process of deployment is influenced by infrastructure constraints, hardware and networking layout, reliability and performance

considerations etc. In this step the actual efficiency of the system in its environment is determined. The final wirings and configurations of the components are made and adapted to the chosen middleware and network infrastructure; furthermore the component execution life cycle is determined. How and when are components initialized? How many instances and replicas of components are present? How are components modified, reconfigured and shut down?

An important consideration that often requires adaptation on the side of the deployed components is in case of dynamic middleware and network infrastructures. In traditional systems, the network layout and architecture is static and does not change during the runtime of the system. In the emerging field of mobile computing, for instance, execution environments, networks and location of components – sometimes called agents in this domain – are not determined and change over time. Such applications need to distinguish functionality and state from the environment and infrastructure they are running on.

Services in the context of deployment and run-time. During the execution and run-time of distributed systems, services can be used as the distinguishable, modular, executable, deployable entities that make up a service-oriented architecture, cf. Figure 2. Services can be instantiated, initialized, registered and published, looked up, accessed, deployed and dynamically connected.

Important in flexible service-oriented architectures is the capability to dynamically look up services during run-time. A central registry or search site needs to be common knowledge in such a system landscape. Services that match certain syntactical criteria and fulfill certain behavioral assumptions can be selected and used immediately. This collaboration is often called the service triangle based on the three roles involved (shown in Figure 3; see Figure 4 and Section 3.1 below for more details). Web service infrastructures, for instance, often contain requester entities, provider entities and registries that communicate using standard protocols such as SOAP for message exchange, WSDL for service description and UDDI for service registration and look up.

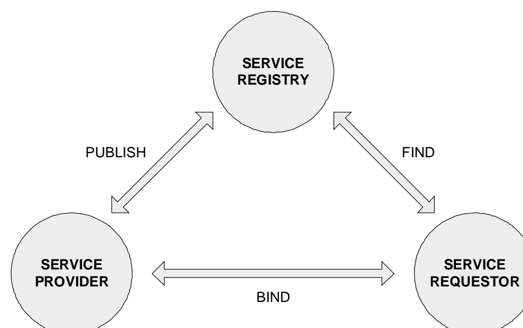

Figure 3: Web Service Basic architecture

In mobile environments, services are the pieces of functionality that networks and mobile applications make available to the user. Availability of certain services is subject to a certain context or location and the service itself might require adaptation depending on the current virtual or physical environment. The environment itself might change, as might the location of the user relative to the environment. Systems need to reflect that, and adapt, reconfigure, recalibrate etc.

Benefits. Services provide a suitable level of abstraction to deploy independent pieces of functionality on network infrastructures and service-oriented middlewares. Service registration, lookup and invocation can be standardized and thus it is possible to created dynamically adaptable service-oriented architectures. In mobile environments, the abstraction of a service enables dynamic applications that adapt to a user's or device's environment.

In case a service-oriented architecture is completely defined on a logical level of abstraction [18], the actual deployment of services onto target components can be left to the middleware. The middleware then will take care of all related infrastructure issues, the initialization and provisioning of the services and the internal wiring and communication links.

3. Service-Orientation in Specific Application Domains

In this section, we present an overview of the use of services and similar notions in specific application domains. Business information systems, for instance, are strongly influenced by service-oriented architectures, web service related standards and middleware. However, the ideas of services or service-oriented development have already been applied much longer ago for telecommunications systems under the name features and feature-oriented development

In the following, we present service notions within application domains and identify prevalent terminology and concepts. We begin with business information systems (BIS) and the web service domain; these currently form the largest sub-community within the service-oriented community.

3.1. *Business Information Systems and Web Services*

Business Information Systems and Network/Internet enabled systems are mostly reactive in nature and provide a number of functions or services to the environment to support business processes and organization goals. This often involves complex data manipulations and legacy application integration. The advent of networks and distributed resources, requirements of scalability and parallelism, and of reliability and replication lead to complex distributed systems.

Such systems are difficult to develop. Often, very heterogeneous environments and system landscapes exist in organizations where each piece has a defined purpose. For instance, an enterprise information backend (e.g. SAP) provides customer relationship management (CRM) functions, a middle-tier application manages the order handling for an online store and web servers provide the web front-end for end users. Data for specific business functions, such as processing a catalog-based, personalized order through the online store is distributed over different locations: the enterprise backend, the order system database and a general document archiving solution.

Integrating such applications running on servers of different hardware, operating systems, middleware frameworks, networks, programming languages, data formats, with different levels of performance, throughput, load over time, responsiveness, reliability and security guarantees etc. is very tedious and error-prone. The high heterogeneity requires a loose coupling of the different applications and integration techniques based on open, widely accepted standards.

Web services [2], web service-enabled middleware [1] and service-oriented architectures have become a major driver in innovation and technology for systems integration. The general ideas of the web service approach are not new. It effectively combines message-driven applications, distributed object systems and open Internet standard applications. Proven ancestor technologies include CORBA [5], DCOM, the World Wide Web, the Java Enterprise Platform [6], and messaging middleware. But only the combination of those technologies and approaches, the foundation on accepted easy to use standards and the behind driving force by several major players in the IT industry give it a big leverage. Web service integrated systems provide loose coupling of heterogeneous systems including legacy systems and thus create stable extensible, scalable architectures.

3.1.1. *Basic Concepts*

The Web service architecture [19] defines that web service systems are built upon 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 and usage interface. 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, a *mediating middle instance* (registry, directory), which connects service requesters and service providers; it provides service registration and lookup functions to both providers and requesters. Service providers themselves can act as service requesters and thus form higher level services by aggregating or composing other services. An interaction between service requester and service provider is often called a *conversation*. Figure 4 depicts these concepts (WSD stands for web service description).

Web-service systems or service-oriented architectures (SOA) are loosely coupled systems composed out of individual components or subsystems that are integrated and connected via web services. The focus of service-oriented architectures is the provisioning of individual versatile, extendible, reusable services that contribute to the business goals of the system's users. Providing such first-class services, and the ability to discover them, is the driving factor in designing the software architectures of such systems; thus by integrating them, loosely coupled systems that are highly scalable, robust and extendible can be created.

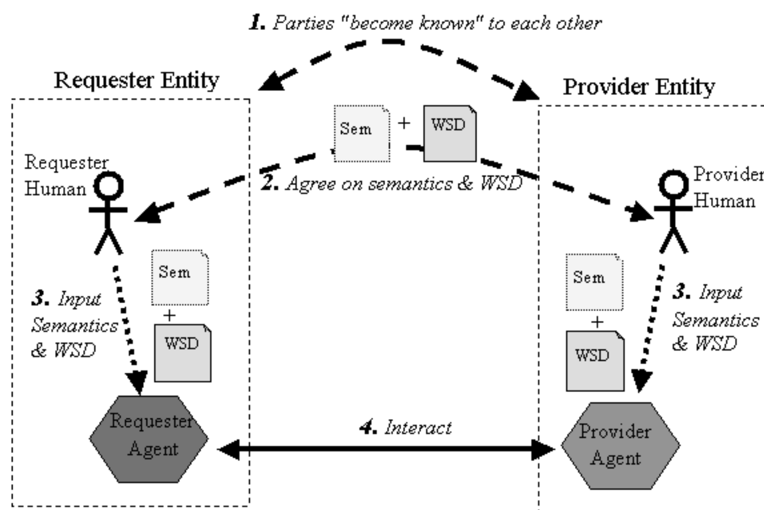

Figure 4: Web Service interaction (from [19])

Web service architectures focus on interactions between distributed entities based on document exchange. An XML document, as delivered by the requester to a service provider constitutes the sole input for the service provider; XML is used here as human readable, platform independent instrument of communication. Neither the way how the document travels between service requester and service provider, nor the way the result document is created is specified and relevant to the web service system. Thus, web service systems emphasize a *document-centric view* on distributed architectures that shares many commonalities with messaging infrastructures. Viewing web service architectures as a new, polished, simplified form of distributed object communication does not comply with the real intention of web services, namely loose coupling of asynchronous processes based on simple, document-centric designs [20].

Although service-oriented architectures do not require or emphasize specific technical infrastructures, specialized middleware frameworks exist. It were these frameworks that made the early, widespread success of this approach possible. Today, middleware application or web server frameworks, such as DotNET [1] and J2EE [6] often provide

the infrastructure to encapsulate self-contained component functionality and provide it as web services in a network where these services can be discovered and used. Tool support further automates development of such systems in never seen-before ways. It is basically a matter of minutes to define, deploy, publish and access existing component functionality, made accessible through web services via Internet technologies, such as HTTP on web servers.

3.1.2. *Definition of Web Services*

Web service related approaches and technologies are still maturing. Supporting standards and techniques are forming and tool support is crucial. Different vendors follow different strategies in marketing the web service technologies, which often originate from existing product portfolios of certain companies. Thus, a comprehensive, concise, commonly accepted definition of web services and service-oriented architectures is still missing.

The W3C glossary [21] gives definitions of the term web service and related terms. We follow the definition of [20] and define a web service as follows:

> **A web service is software that processes input in form of XML documents and produces output in form of documents. The implementation of the service is not defined or relevant – neither is the transport medium or its characteristics. The input and output documents follow an agreed upon XML schema that is part of the service description. The documents are the only input or state information a service has and the output is the only result a service produces. Web services are bound to accessible points in a network.**

Web service interactions can be arranged to form conversations. This process is called service combination, composition, aggregation, choreography, or federation.

Systems that provide web services to the environment and/or that are composed out of web services are called service-oriented architectures. The basic principles that apply for individual web services (e.g. small set of simple and ubiquitous interfaces to all participating software agents, only generic semantics are encoded at the interfaces, schema constrained messages for information exchange, no system behavior prescribed by messages, extensible interfaces) hold true for the entire system as well.

A commonly seen application of web service technology is SOAP RPC interactions. Remote procedure calls (RPCs) are encoded using web service interfaces and protocols. This is a form of distributed object communication. SOAP RPC web services do not fulfill the above definition, because interface contracts prescribe both system behaviors and application semantics. System behaviors are difficult to prescribe and thus these services are not really interoperable.

In order to make use of web service infrastructures and architectures, web services need to be described in common, agreed upon, computer interpretable exchange formats. Web service descriptions are used to register web services in directories and registries and to make the web service interface accessible for service requesters. The standard way to describe web services is using W3C's WSDL [17] format, an XML based language for describing network services based on abstract message exchange descriptions between endpoints realizing operations [22]. Web service registries and directories keep track of web services, accessed by service providers and service requesters.

To form applications systematically out of a set of individual web services, some notion of service composition is required, based on service models facilitating composition. Several approaches and standards exist for web service composition, such as Business Process Modeling Language (BPML), WS Choreography Model, and Business Process Modeling Notation (BPMN).

3.2. *Telecommunication Systems*

Many roots of service-orientation lie in the area of telecommunication. In this section we will give a short introduction of probably the most prominent approach of this domain: the one of Zave and Jackson [4][27]. The approach is very domain-specific in its nature and mainly focuses on modeling routing problems and feature interaction.

For telecommunication systems it is assumed that a base of functionality already exists.[a] The main goal of feature-oriented approach is how to add, remove, modify, and combine pieces of functionality later in the life cycle of such systems.

A **feature** (of a software system) is an "optional or incremental unit of functionality" [27]. The feature specification contains an action, enabling condition, and priority. The action is performed if the enabling condition is true and the feature has the highest priority. 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". The set of possible system behaviors is obtained by applying a composition operator to the base description and the feature descriptions. The composition operator must ensure that in any situation the feature with the highest priority must be performed. In case of features with a lower priority, their enabling conditions must be changed accordingly [27].

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** which is defined as "some way in which a feature or features modify

[a] This might be an existing implementation of a POTS (Plain Old Telephony System) or any other system.

or influence another feature in describing the system's behavior set" [27]. Zave and Jackson explicitly distinguish between wanted and unwanted feature interaction. Unwanted results of feature interaction are incompleteness, inconsistency, non-determinism, and unimplementability. Desired feature interaction "can be" achieved without changing any feature modules, rather by simply adjusting the precedence relation; i.e. the order in which feature can occur in a route. Desired feature interactions occur for instance when there are shared resources across features due to hardware resource limitations. All kinds of interactions however indicate dependencies that need to be specified and tracked to ensure a consistent system.

Assumed, that (parts of) a system already exists, new features are added subsequently by performing the following steps (see [4]).

(1) Describe new features as if they were independent (manually)
(2) Identifying and understand all potential interactions (help of automated analysis necessary)
(3) Classify interactions as bad or good (manually)
(4) Adjust feature descriptions so that the result contains no more bad interactions

To realize this process, a modular formalism is needed. For that purpose, Zave and Jackson introduce the Distributed Feature Composition (DFC) reference architecture which is a feature-oriented virtual architecture for specification and implementation of telecommunication services. DFC is highly domain-specific. It was designed with the following specific goals: generality, analyzability and behavioral modularity [28].

DFC is a virtual architecture with the basic goal to support usages. A usage is a dynamic assembly of features, line/device interfaces and internal calls to satisfy systems requirements to connect two end points on behalf of a user with a given set of requested features. A router component is responsible for establishing a connection between the end points across the source and target zones, activating a certain number of features along the call. All features have a certain activation protocol and can be invoked independently from each other. Features are connected to each other and to end points by internal calls. To avoid or control feature interaction, all features have a priority that determines the order in which they can be applied by the router. The basic idea behind DFC features is that they are autonomous, modular units that can be applied in sequence by adhering to a standard black-box feature interface. In this way, the DFC architecture follows the "Pipes-and-Filters" architectural style described in general in [29].

The router is an essential element of DFC. It is a virtual component that routes calls to target addresses and applies features as requested. The router uses data on feature subscriptions, feature precedence, and other configuration to perform the routing. The router can access global operational data and configuration information (e.g. a user's

forwarding address) only through an application of features. This ensures full feature modularity and data partitioning. Figure 5 depicts the components of the DFC architecture with the router as a central element.

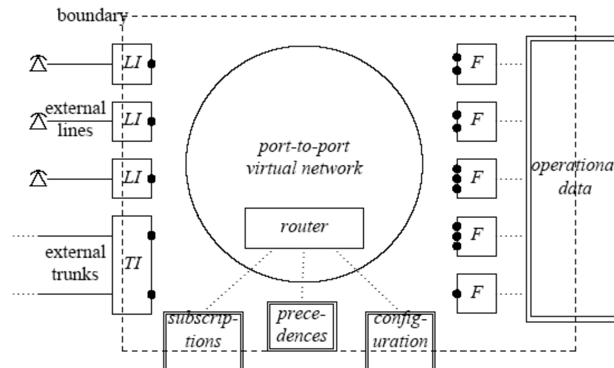

Figure 5: Components of the DFC architecture [4]

The precedence relation defines the order in which features occur in a route. It is the only place of the algorithm where feature relations are captured. The goal is to place features with a higher priority later in the route, close to the target zone. The proponents of the methodology claim that their concept of features and feature precedence provides a useful degree of behavioral modularity and that it is possible to manage desired and unwanted feature interactions by simply adjusting the precedence relation without modifying any of the feature modules themselves.

4. Service-Oriented Approaches at TUM

In this section, we describe various service notions developed at the Software & Systems Engineering Lab at the Technische Universität München. Most of these approaches come from a practically motivated research background out of industry collaborations and have a strong formal foundation.

4.1. *The InServe Methodology*

InServe [8] is a recent research projected at the Chair of Software & Systems Engineering. It is funded by the Deutsche Forschungsgemeinschaft (DFG) [30]. InServe focuses on researching and establishing a service-based development methodology based on a solid formal foundation. This includes research on description techniques, processes, service-oriented architectures, as well as foundational research for a comprehensive service notion and service composition. Furthermore, InServe provides prototypical implementations to support the concepts and conducts several case studies to evaluate the

results. InServe is joint work by a collaboration of the groups of Broy and Krueger [31] and combines concepts of both groups [9].

The service notion is very prominent in InServe as a concept for modeling and design entities. Based on a structured requirements analysis, systems are defined and designed in terms of services.

> **Service Definition:**
> **Services are distinct, precisely defined units of functionality that the system or one of its components offers. Services are defined in terms of interactions between entities of the system and its environment. This notion of service thereby models system structure, state and behavior that pertain to certain functionality.**

In an iterative process, the system (or one of its components, respectively) is associated with a global state model and can be structured into a set of logical entities. Logical entities are connected via communication channels. The system state can be refined into local states of the structural entities. A selection (combination in form of a predicate) of logical entities and states forms roles. Services are defined as interplay of roles to provide certain functionality. Services can be annotated with meta-data to contain Quality-of-Service properties or other information.

The service definition and the logical system architecture that is implied by the service repository do not impose any restrictions on the actual technical implementation of the system. Roles can be mapped flexibly to components to form deployment architectural configurations. This mapping can happen during design time by a manual configuration step of a system designer, during execution time by automatic configuration of a powerful middleware, and as a combination of both. In the latter case, the system designer specifies the technical constraints, for instance the required existence of certain computational nodes, ECUs or legacy systems, and the middleware performs the remaining mapping to satisfy the constraints and QoS properties that were specified.

Service elicitation and architecture mapping are performed in an iterative development process, see Figure 6. The process can begin at any step and changes can be made at any point in the process. The process supports iterative design, refinement, refactoring of reactive systems with services as first level elements.

As shown in Figure 6 (left side), part (I) of the process consists of a structuring of requirements in form of use cases, non-functional requirements including QoS constraints and other constraints for the system. Based on these structured requirements, a common repository of structural entities, associated state models, role definitions and service specifications is created and can be iteratively refined.

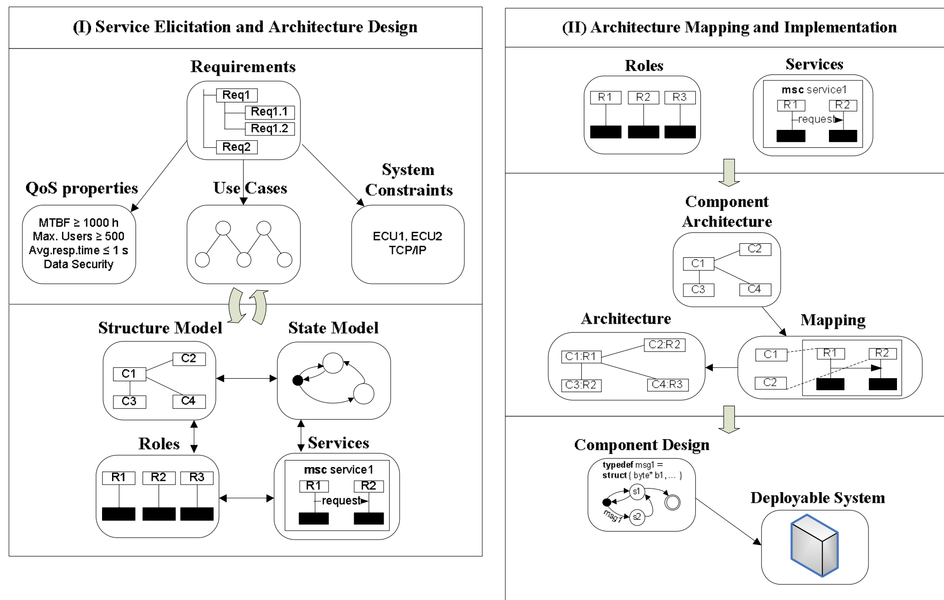

Figure 6: Process phases (I) and (II)

In phase (II), depicted in Figure 6 (right side), the elements of the logical architecture and design model, in particular roles and services, are starting points to elaborate a component architecture. Roles and thereby the system services are mapped to the architectural components. This mapping step realizes the separation of the logical service-oriented behavioral model and a specific technical deployment architecture. Service model and architectural description together build the basis to generate executable implementations of the system.

Applications of this model-based approach for the development of service-oriented systems are efficient architecture exploration and generation of executable prototypes from logical service-based behavior models [32][16], conformance testing of binary components against the executable specification, model-checking of the logical model, architecture validation, and product-line development support by mapping of one core service repository to multiple deployment architectures.

InServe provides tool support to specify services, their dependencies and target architecture mappings, so that for instance code generation steps can be automated directly from that model. The SODA tool (Service-Oriented Development and Architecture Design) [32] governs the described development process and realizes a tool

platform. Tool adapters are planned to the AutoFocus [33] and AutoRaid tools and to the tool suite of the S3EL team at the UCSD [32].

4.2. *MEwaDis*

MEwaDis [9] was funded by the Bayerische High-Tech Offensive and was running from July 2004 to May 2006. Collaborations include the Chair of Software & Systems Engineering (Technische Universität München), the BMW Car IT, BMW F&E and the ESG. The primary goal of the project MEwaDis is the development of techniques for the analysis, modeling and validation of reliable, adaptive, context-aware services, and of process models for their development [36][37]. The results are prototypically implemented in the domain of automotive systems.

The notion of service in MEwaDis is rooted in requirements engineering but influence also early phases of system design and architecture. Services are black-box descriptions or placeholders of functionality. More formally, services get input via channels from explicit sources (users, other components) and implicit sources (the environment) and produce output. The environment is modeled externally in form of an environment model that provides state and reactive stimulus for the service. Services can be connected to form service networks.

Services are related to each other with different kinds of relationships [11]. Services can interact with each other, which can be expressed even more precisely after a refinement as a controlling or calling or monitoring relationship. Or services can be defined as being independent from each other. This precise definition of service relationships allows for a dependency and completeness analysis of the service or functional model of the system. It is possible thereby to detect feature interactions or potential problems due to close relationship between certain functionalities (e.g. in a security analysis [38], by model-checking). Furthermore, if the service network is strictly refined into a design model, certain service relationships translate into certain architectural or design patterns that provide certain properties in the system implementation. Independence of certain functionality in the system can thereby be guaranteed.

4.3. *Services in mobilSoft*

In the mobilSoft project [9], Bavarian car manufacturers and suppliers together with four research institutes work on a *software development process for automotive embedded software*. The intermediate results presented in this document are currently further elaborated in the course of the project.

During the development of complex (embedded automotive) systems many aspects have to be taken into consideration. For example, the system functionality, the logical system structure, the technical realization. Therefore, it has proven to be useful to base a development approach on *abstraction levels*. The system under consideration is described

by a set of domain specific abstraction levels (in the following also called *modeling levels*) which are built upon each other. Hereby, the modeling levels (respectively) provide self-contained concepts for the representation of the information, which are specific for each level. In the mobilSoft project both the requirements engineering process and the design process are based on abstraction levels.

In the course of a model-based requirements engineering phase, the requirements are (partially) formalized and specified by means of models. Hereby, the kind of requirement – e.g. functional, structural, technical requirements – determines the type of model that is used to specify the requirement. The problem is that although at the end of the analysis phase, the requirements are formalized by means of model fragments, they are not or only loosely related to each other. It is difficult to identify unwanted interactions or contradictory requirements.

Therefore, on the most abstract level of the design abstraction levels, the *service level* is introduced. On this service level, the majority of the *functional* requirements are formalized consistently and related to each other in order to detect contradictions and interactions. The aim of the service level is the consolidation and precise specification of those requirements that describe the black box system behavior. Here, black box system behavior depicts the behavior that is visible for the user (which does not only refer to a human user but also to another system). Figure 7 shows this process graphically. We see the service level as *the* interface between the requirements engineering phase and the architecture design phase.

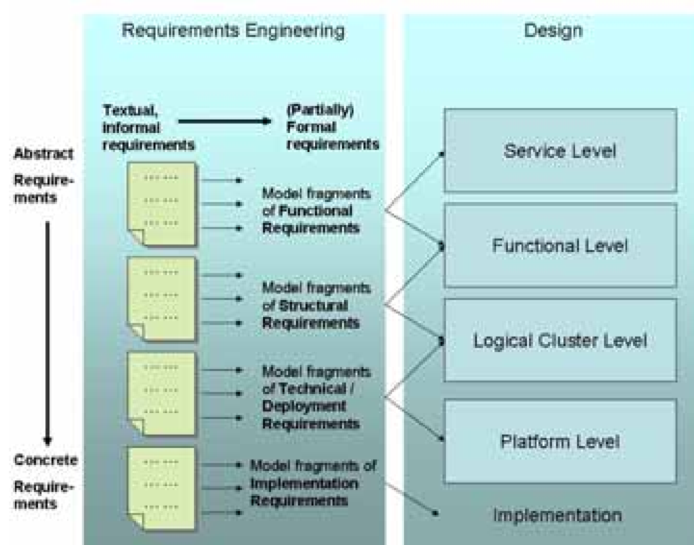

Figure 7: Abstraction Levels in the mobilSoft project (work in progress)

Services describe the black box system behavior; i.e. hat means, *functional behavior* that is visible to the user. A service describes a partial behavior by a black-box specification of the system. Latter is done by a mapping of input to output actions. Logical actions abstract from technical signals or messages. However, the representation of data (how it is processed by the realization of the system) is not under consideration. Only abstract data types are used.

The project description can be found in [39]. For more information about model-based requirements engineering in mobilSoft please refer to [40]. The role of services in the mobilSoft approach is described in [41].

4.4. *Functional Architecture Modeling*

In this section we describe the approach of Schätz et al. [42][43][44][45]. The main focus of this approach lies on the explicit modeling and specification of functional architectures of distributed reactive systems. The presented approach has the following basic idea: First, different functions of a system are specified in a modular way. Later these functions are integrated into a functional architecture. The step of functional integration makes use of well-defined, formally founded operations to ensure compatibility and consistency of the functions. The approach uses the concept of services to encapsulate pieces of functionality. The methodology puts the *functional integration* already at an early stage in the development process.

Services are the modular pieces of functionality that are used to describe the functional architecture of a reactive system [43]. Examples for services in automotive systems are “Window Up” and “Child Lock Off” of a power windows control software (cf. also [43]). A service is defined by its interface, its variables, its configurations, and its transitions. **Variables** (e.g. counters, flags, values) are used to describe the data space of a service. The **interface** of a service describes the input ports monitored by a service and the output ports controlled by a service. Services communicate by exchanging signals over typed and directed **channels**, connecting input ports to output ports. Thus, a **port** is a typed end point of a channel and is part of the interface of a service. A port can have a value received via a channel. The architectural aspects of a service are defined by its input ports, its output ports, its variables as well as its control locations.

The behavior of a service is described by a collection of configurations and transitions between configurations. A **configuration** describes a certain state of the system. A configuration itself may be atomic (e.g., Init, Up) or contain a collection of sub-services active in this configuration (e.g. Child Lock Off). A **transition** describes how a service changes from one configuration to another. To provide a methodological handle for such a state transition, **connectors** describe the entry (e.g., Start, Off) and exit points (e.g., On) of a configuration, defining how a configuration is activated and terminated. A transition

thus links exit connectors to entry connectors. The computational model of AutoFOCUS is used for the specification of reactive behavior [1].

The Functional Architecture Modeling approach uses the following notions [43] to describe the behavior of a service: A **state** is an assignment mapping input and output ports as well as variables to their current values. A **step** is a pair of states describing the change between the two states, called the pre-state and the post state. An **observation** is a either a finite or an infinite sequence of states with a given starting location and – in case of a finite sequence – an ending location. During the execution, a number of actions are performed that change the date state as defined by the steps. The **behavior** of a service is the set of observations performed by the service. Furthermore, since services are focused on modeling partial behavior, the semantics introduced here explicitly specify undefined behavior, either in form of undefined values of variables or ports, as defined above, or in form of partial executions. As far as notation techniques are concerned, so-called service configuration diagrams are introduced to graphically describe services as well as their configurations [43].

By using *disjunctive combination*, services are combined to form alternative configurations. The disjunctive combination of two services results in a service that makes use of the combined input and output ports as well as variables of each service, accesses the combined control locations of service, and exhibits the combined behavior of each service.

The *conjunctive combination* of two services results in a service that makes use of the combined output ports, variables of each service, the combined input ports of each service, accesses the joint control locations of service, and exhibits the joint behavior of each service.

Atomic configurations are the basic building blocks of behavioral description and have no internal structure. They only have an interface consisting of input and output ports, entry and exit points, and their behavioral description consisting of configurations, connections, and transitions. Combining services to configurations basically corresponds to a simultaneous activation of these services. Services are composable, if each entry connector of the configuration is simultaneously linked to an entry connector of each service; and each entry connector of a service is linked to an exit connector of the configuration, simultaneously with an exit connector of each service of the configuration.[b]

---

[b] Since service construction is focused on the combination of functionality rather then the composition of components, several services may simultaneously access the same port. This is possible by mapping ports of different services to the same port of the configuration. If these services disagree on the value assigned to the port, the value of this assignment is inconsistent.

The approach offers two techniques for the combination of specification modules, joining and merging. When joining specification modules, each specification module describes an independent part of the behavior of the component under development (disjunctive combination). When merging specification modules, each specification describes the same part of the behavior of the component concerning a different aspect (conjunctive combination).

4.5. *The FOCUS/JANUS Approach*

A formal theory of distributed systems and services is the JANUS approach [15]. JANUS is based on the FOCUS theory [46]. The aim of the FOCUS theory is to first specify systems as families of components (and their interfaces) and to then put the components together (*composition*) forming the architecture. The verification of an architecture (prove of the interface specification of the system) then can be achieved by the interface specification of the components and the composition verification rules.

The main goals of the JANUS approach are to give a formal model for services, layers, and layered architectures. Furthermore, a theory for relating, composing, and refining services, layers, and layered architectures is aimed at. Advanced goals are specification and verification techniques, a methodology for designing services and respective architectures, and design patterns for services, layers, and layered architectures.

In the following, we summarize the main concepts of the FOCUS approach with the JANUS extensions for service-oriented development. Informally, in FOCUS [46], a system is composed of a number of components which encapsulate behavior (principle of *modularity).* We face a *relative notion of system* here, i.e. components can be composed to components, again (“A system is a component is a system” [47]).

Interfaces exist between components and between components and the environment. The communication takes place via directed communication channels. A mapping of message streams to channels represents the exchange of typed messages over these channels in an asynchronous way assuming a system wide global clock. The behavior of such systems is expressed as equations or relations relating a number of (possibly infinite) input sequences to a respective set of output sequences that are resulting as part of the input. More precisely, the communication is expressed in terms of relations on streams; streams represent histories of communications of data messages in a time frame. Multiple possible outputs for a certain input express non-determinism. Furthermore, the channels are divided in disjoint input and output channels for a component, respectively.

Causality of such a system is enforced by requiring that every computed output is only dependent on inputs of the time slots strictly before the output happens.

The behavior of components is described by their black box behavior, i.e. their interfaces. The interfaces provide a *syntactic* and *semantic* notion. The syntactic interface associates a type for the component whereas the semantic interface describes the observable behavior. Let *I* and *O* be sets of typed input and output streams, respectively. *(I,O)* denotes the *syntactic interface* of a component. It defines the types of messages that can be exchanged by a component. The *semantic interface* with the syntactic interface *(I,O)* is represented by $F: H(I) \rightarrow \wp(H(O))$ that fulfils the *timing property*. **A component is a *total behavior*** (see proceeding paragraph). Figure 8 shows a graphical representation of a FOCUS component.

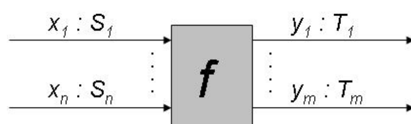

Figure 8 Graphical representation of a FOCUS component with input channels $x_1, \ldots x_n$ and output channels $y_1, \ldots y_m$ and their respective types $S_1, \ldots S_n$ *and* $T_1, \ldots T_m$

The timing property demands that the set of possible output histories for F for the first *t+1* only depends on the inputs of the first *t* time intervals. (The processing of messages takes at least one time interval.) Functions that fulfill this timing property are called *time-guarded* or *strictly causal.* The application of a strictly causal function leads to either an empty output for all input histories, or a non-empty output for all input histories. In the first case, we call the function *paradoxical*, in the second case we call it *total.*

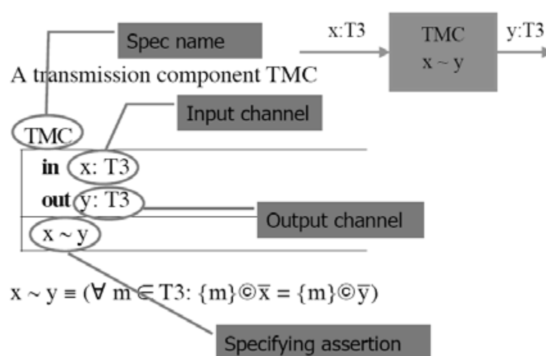

Figure 9: Example for a component specification in FOCUS [47]

Fehler! Verweisquelle konnte nicht gefunden werden. gives an example for a simple component specification [47]. A transmission component (with the name TMC) is specified. It has the input channel *x* with data type *T3*, and the output channel *y* with the same data type *T3*. A structural view on the component can be seen in the right upper

corner of **Fehler! Verweisquelle konnte nicht gefunden werden.**. The specifying assertion is *x~y* which expresses that each data on the input channel is transmitted on the output channel.

In the following we will see, how these basics are used in the JANUS approach to define service, layers, and layered architectures.

Component interfaces provide functionality. The idea is to describe for each piece of functionality under which conditions (*precondition*) the functionality may be invoked and which affects this has (*postcondition*). This leads to the notion of service.

A service is a *partial behavior* as opposed to a component which is a *total behavior* (see above). Partial means that a service is defined only for a subset of input histories. This subset is called the *service domain*. The output of a service is called *service range*. A service basically is a set of interaction patterns. A service fulfills the timing property only for the input histories with nonempty output set. A service gives a partial view on the behavior of a component. To be able to use a service, certain access conventions must be valid (*service protocol*).

Example. Figure 10 gives an example for a simple service specification using the same specification scheme as above. This time a queue is defined which reads in some data and gives it out on demand. Note that this specification is *partial* as it does not say what to do when data is requested in case the queue is empty.

type QIn = {req} ∪ Data

type QOut = Data

Queue
in x: QIn **out** y: QOut
$\{req\}\#x = Data\#y \wedge \overline{y} \sqsubseteq Data©\overline{x}$

Figure 10 Example of a simple service specification in the JANUS approach [47]

Services are partial functions. Therefore it makes sense to put special emphasis on the characterization of the service domain when specifying services. Both input assumptions and output commitments are specified. Input assumptions control if some input is valid for a special service. As the conformance of input histories may also depend on previous outputs, input assumptions are specified by predicates on both the input and the output histories.

In a layered architecture, each layer provides an upper interface (*export interface)* and makes use of a lower interface (*import interface*). A *service layer* is a layer the syntactic interface of which is structured into two (or more) complementary sub-interfaces. Note that the behavior of a service is not changed, i.e. the service remains the same from a behavioral view. Only the interface is divided into two sub-interfaces. The composition of several layers results in a layered architecture. **Fehler! Verweisquelle konnte nicht gefunden werden.** depicts this.

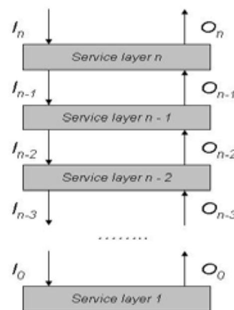

Figure 11 Layered architecture in FOCUS being comprised of a set of service layers [15]

For more information of the JANUS approach and the FOCUS theory please refer to [15][46].

5. Comparison and Evaluation

In the previous sections, we have described several service-oriented approaches. In the following we compare these using criteria that we introduce and describe below.

In Section 5.1, we describe the evaluation criteria according to which the comparison will be done. We explain each criterion briefly and discuss its importance for service-oriented approaches and possible trade-offs. In Section 5.2. , we give a tabular comparison of the different service-oriented approaches using the introduced criteria.

5.1. *Evaluation Criteria*

We have shown that services and service-oriented development approaches occur in many forms throughout many different application domains. Comparing such different concepts and approaches requires thorough analysis and domain-specific knowledge. The main question is: *According to which criteria should the service-oriented approaches be compared?* In this section we will list evaluation criteria that we refined from analyzing the different approaches and that play an important role in each of those.

The evaluation criteria that we will use are the following:

(1) Coverage of development cycle
(2) Functionality-oriented development
  (a) Specification of functional properties
  (b) Capturing functional dependencies and interactions
(3) Capturing global and local views on services
(4) Abstraction from system structure and deployment infrastructures
(5) Black-box service interfaces
(6) Capturing cross-cutting concerns
  (a) Description of Quality-of-Service properties
  (b) Specification of exceptional behavior
  (c) Specification of forbidden behavior
  (d) Support of data flow, transactions and data integrity
  (e) Support of security properties
(7) Support of partial specifications
(8) Support of under-specification (non-determinism) and refinement
(9) Methodological and process support (steps, refinement, model transformations)
(10) Service description and discovery mechanisms
(11) Runtime support, middleware, infrastructures
(12) Support of dynamic execution environments and adaptivity
(13) Degree of standardization and community-acceptance
(14) Degree of formality and precision
(15) Level of direct applicability or generality
  (a) Direct applicability and degree of practical support
  (b) Generality and domain-independence

We consider the evaluation criteria (1) to (6) be the **significant characteristics**. In the following, we explain the evaluation criteria in detail.

(1) Coverage of development cycle. The software development process can be partitioned on a logical level into consecutive phases. We consider the phases

(a) Requirements elicitation, analysis and specification
(b) Architecture modeling and design
(c) Fine-design, implementation and coding
(d) Integration and verification
(e) Deployment and configuration
(f) Run-time and execution

Different approaches have different emphasis. In Chapter 2. , we have shown the benefits of providing methodological support for certain phases. The more phases an approach supports, the less conceptual gaps and potential inconsistencies will occur,

leading to more seamless, intuitive development. Additionally, combining multiple approaches will lead to a higher overhead, complexity and reduced traceability of results.

(2) Functionality-oriented development. The development of today's complex multi-functional systems, found for instance in telecommunications or the automotive industry, requires techniques that emphasize functions, functional properties and functional dependencies rather than structural units, such as components. Emphasizing function requires a higher level of abstraction and a closer integration with requirements engineering. We evaluate service-oriented development approaches by their support for

(a) Specification of functional properties
(b) Capturing functional dependencies and interactions

Criterion (a) determines if the models, notations and processes of an approach support the explicit specification of different functions, features or services of a system, separately from each other. Criterion (b) indicates whether functional dependencies can be specified, which are often caused by complex interactions within a system or between a system and its environment. Thereby, approaches can handle interaction complexity and detect unwanted feature interactions early in development on a logical level, thus reducing costly system errors and increasing system maintainability.

(3) Capturing global and local views on services. Multi-functional systems often require in interplay of independent services. Helpful in development are models that provide both views:

(a) Local view, showing one service in isolation
(b) Global view, showing the interplay of multiple services

Local views emphasize the individual service as modular building block with its syntactical and behavioral interface; they show how a service acts as a role in a conversation, the service protocol. One specification style for instance provides pre- and post-conditions in form of assumption/commitment specifications. Service users can check the pre-conditions before they use the service. Global views focus on the interplay of services in order to show and realize higher-level system services, typically by giving interaction choreographies. Understanding "the big picture" is most significant when designing multi-functional systems. Local and global views on services complement each other: locally specified services can be refined into and implemented by an interplay of other services, modeled in a global view. Thus, local and global views provide black-box and white-box views on services, and in combination result in a consistent understanding of a multi-functional system, and a scalable hierarchical service-oriented development process.

(4) Abstraction from system structure and deployment infrastructures.

The behavior of a multi-functional system is independent of its implementation and structure. Approaches that separate behavior models from system structure and deployment infrastructures support the development of long-lasting, reusable, high-value function repositories that can be applied to multiple possible and changing implementations. Through this abstraction, they also reduce complexities related to the combination of behavior and structure. Models on different levels of abstraction capture different aspects of system design: Functionality and functional dependencies on a logical level, and implementation details on lower technical levels. Structural information, such as distribution over components, classes or other computational entities, can be added by refining the functional model. Alternatively, the system functionality can be mapped to a structural model (architecture) of the system. This abstraction also supports software product line engineering, because variants of the common functional repository can be mapped to similar or different target architectures.

(5) Black-box service interfaces.
Service interfaces that provide only syntactic and semantic specifications of a service realize a black-box service interface. Any information of how the service is provided is hidden. Many possible implementations of the service can be provided and exchanged over time. This abstraction from implementation increases maintainability and realizes a higher degree of decoupling.

(6) Capturing crosscutting concerns. Cross-cutting concerns usually capture non-functional requirements that can be associated with services and other aspects that apply to the entire service. Service-oriented approaches differ in the degree of support for the specification of crosscutting concerns, non-functional requirements, and in the degree of formality. We evaluate the following sub-criteria:

- (a) Description of Quality-of-Service properties
- (b) Specification of exceptional behavior
- (c) Specification of forbidden behavior
- (d) Support of transactions, data flow and data integrity
- (e) Support of security properties

Description of Quality-of-Service properties. Quality properties of services are a central part of non-functional requirements attached to services. Examples are performance and timing requirements, availability, robustness, etc. Quality properties are often a distinguishing factor when choosing from multiple available services with similar functionality.

Specification of exceptional behavior. Some approaches enable the specification of exceptional behavior. Exceptional behavior is supposed to happen in case pre-conditions of a service execution are violated or some errors have occurred. Exceptional behavior specifications define how to handle the exceptional situation and potentially how to roll back to a consistent state of execution. Doing so ensures an overall consistent cross-

cutting treatment of exceptional cases and thus data and system safety and integrity. Handling exceptional situations consistently throughout the system is very difficult on the entity (component) level. Fall-back and recovery strategies commonly span the interplay of several system entities.

Specification of forbidden behavior. A system must never exhibit forbidden or negative behavior, as specified by some approaches, thereby complementing any positive specifications. Being able to specify negative behavior leads to much more compact and understandable specifications, because the number of forbidden behaviors can usually be kept small, and regular behavior uncluttered.

Support of data flow, transactions and data integrity. Data flow aspects are significant for many applications and therefore need to be considered besides structural and interaction concerns. Transactions associated with a service coordinate the resources that are involved in providing the service, ensure data integrity and provide defined checkpoint and roll-back strategies in case of resource conflicts or unsuccessful operations within a transaction.

Support of security properties. Security properties and requirements need to be associated with service definitions in domains where security is a special concern. This includes privacy, authenticity and access control aspects. Security, as important cross-cutting concern needs to be considered from early on in the development, designed into the system and verified against the specification.

(7) Support of partial specifications. Partial specifications apply only to subset of all possible inputs and conditions. Methodically they are important to avoid the need of providing complete (total) specifications early on in the design process. Independent, partially defined pieces of behavior can be combined later, for instance after applying a closed world assumption and checking for consistency. The condition that defines the validity of a partial specification is the service enabling condition, service protocol or service domain. A service definition does not constrain any other behavior that does not fulfill the service condition. Partial specifications enable for instance to separate regular from exceptional behavior, as discussed above. Service-oriented approaches are well suited as partial behavior specifications, because services can be seen as abstractions or projections of system behavior on certain functionality. Such partial views are less complex and easier to understand and modify.

(8) Support of non-determinism and refinement. Non-determinism exists when a specification enables multiple possible outputs for a certain input. Non-determinism is one form of modeling under-specification. Such specifications leave room for later refinement in case information is not yet available or should be left open. This is methodically important to support a smooth, seamless development process. Through

refinement, it is possible to add more constraints to a specification to reduce the number of possible behaviors in order to make a specification deterministic and thus implementable. By applying non-determinism and refinement, certain properties of the system can be defined and verified early in the development. Any later refinement operation will not violate any previously proven properties.

(9) Methodological and process support. Methodological support includes refinement and refactoring operations, consistency checks, metrics, model transformations and automated procedures to support the development. Processes embed the methodological operations into workflows that lead from the requirements to a high quality implementation in a repeatable manner, while minimizing the risks for the project. Supporting iterative development instead of pure top-down or bottom-up is significant for the practical applicability of a service-oriented development approach. Formally proven powerful methodological operations increase the efficiency and scalability of an approach and the confidence in the correctness of its results.

(10) Service description and discovery mechanisms. In dynamic systems, service requestors have to find (discover) the services they require. While technical issues can be solved for instance by distributed databases that store and access all available services, the actual semantical discovery (i.e. *knowing* which service is suitable/best for a given problem) remains a challenge. Suitable service-oriented approaches must provide means to specify the characteristics and requirements of a service in a formal way. Then, the discovery process of services can be automated leading to a higher reliability as compared to current techniques of pattern matching / word comparison.

(11) Runtime support, middleware, infrastructures. Significant support in form of technical infrastructures, communications middleware and run-time environments is necessary to realize service-oriented systems in practice. Supporting approaches incorporate such concepts into methodology and process, and provide assistance with implementation, configuration and execution of software. A too specific focus on specific environments or infrastructures reduces flexibility of the approach in general, however.

(12) Support of dynamic execution environments and adaptivity. Describing dynamic execution environments, in which the number and types of interacting system entities or context assumptions that about the environment are changing, requires suitable models and support. Context aware service-oriented systems can take location, network availability and topology, peer communication environment, allowable resources into account when providing and requiring services. Only by dynamic models is it possible to fully specify and verify such systems. Such models must provide abstractions of changing environments and provide means to model adaptivity of the software.

(13) Degree of standardization and community-acceptance. Many standards exist ensuring interoperability and quality levels for specific application domains. They cover topics from process guidelines to implementation level data formats. More conventions and standards exist for modeling, notations, behavior descriptions etc. Approaches can be evaluated by the degree of adherence to standards and the degree of acceptance of these standards in the community. The use of standards and standardized or community-accepted technology and conventions makes practical adoption of new approaches much easier and increases interoperability and formalization.

(14) Degree of formality and precision. Service-oriented approaches offer different degrees of formality and precision. Formally founded service notions can be based on rigid, precise mathematical models that define all notational elements and methodological operations of a system development model precisely. Such models can express system properties in mathematical rigor and provide means to prove them. Clear definitions prevent misinterpretation and enable for instance automatic code generation.

(15) Level of direct applicability or generality. Service-oriented development approaches can be fundamental and theoretically founded, while others may be very practical and provide extensive implementation and run-time support for a certain system class or domain. We evaluate

(a) Direct applicability and degree of practical support
(b) Generality and domain-independence

The necessary effort of tailoring an approach for a specific application domain, platform or technology influences applicability. On the other hand, generality is lost the more technological and notational constrains an approach demands.

5.2. *Comparison Table*

The following table gives a structured overview of how the different evaluated service-oriented approaches compare against each other using the criteria introduced above.

Table 1: Comparison of Service-Oriented Approaches

	Service-Oriented Approach						
Evaluation Criteria	Web Services	Telecommuni-cation Domain	InServe	mobilSoft	MEwaDis	Focus/Janus	Functional Architecture
1.a (covers requirements)	-	+	+	+	+	*	+
1.b (architecture/design)	-	+	+	+	+	+	+
1.c (implementation)	+	+	*	-	+	*	+
1.d (integration/verification)	-	+	-	+	+	+	-
1.e (deployment/configurat.)	+	+	+	+	+	-	-
1.f (execution)	+	-	-	-	+	-	-
2.a (functional properties)	*[1]	+	+	+	+	+	+
2.b (functional dependencies)	*[1]	+	+	+	+	-	*[16]
3.a (local views)	+	+	+	+	+	+	+
3.b (global views)	+	+	+	+	+	+	+
4 (Abstraction from structure)	*[2]	+	+	+	+	+	+
5 (Black-box interfaces)	+	+	+	+	+	+	+
6.a (Quality-of-Service)	*[3]	-	+	*[12]	+	*[14]	-
6.b (exceptional behavior)	*[3]	-	*[8]	*[12]	-	-	*[17]
6.c (forbidden behavior)	-	-	-	*[12]	-	+	+
6.d (data flow, transactions)	*[3]	-	-	-	-	-	-
6.e (security properties)	*[3]	-	-	-	+	-	-
7 (partial specifications)	-	+	+	+	+	+	+
8 (under-specification)	-	-	+	+	+	+	+
9 (methodology and process)	*[4]	+	+	+	+	*[15]	+
10 (description, discovery)	+	*[6]	*[9]	*[13]	-	-	-
11 (runtime support)	+	-	*[10]	-	+	-	*[18]
12 (adaptivity)	*[5]	-	-	-	-	-	-
13(standardizat., acceptance)	+	*[7]	*[11]	-	-	-	-
14 (formality and precision)	-	+	+	+	+	+	+
15.a (direct applicability)	+	*	*	*	+	-	*
15.b (generality)	-	-	+	-	-	+	+

Legend:

+ evaluation criterion fulfilled
- evaluation criterion not fulfilled
* evaluation criterion partly fulfilled (see remarks)

Further explanations:

[1] Web services encapsulate behavior. However, behavior is not specified formally. Dependencies are implicitly specified by web service choreography standards.

[2] On the one hand web services abstract from structure as they encapsulate functional behavior. Technologies can be dynamically mapped to structures (registries, UDDI, discovery, etc.). On the other hand many current standards (e.g. WSDL) explicitly contain structural information.

[3] Additional standards exist which enable the specification of QoS attributes, exceptional behavior, transaction handling, security properties.

[4] There are several approaches under research to enhance web service development with methodology and process. However, there are so far academic and not adapted by the community.

[5] Due to functional abstraction and the possibility of registration and look up, adaptivity is supported in principle. However, adaptivity is not directly addressed in the current standards.

[6] Features are described abstractly and can be discovered and connected dynamically (DFC architecture). However, telecommunication systems are not dynamic in the general sense and features are predefined.

[7] There exist various proven approaches from research and development in this area that are also practically applied. However, these technologies are not generally accepted and there are several competing ones. New systems and standards (IP technology) make it difficult to apply the same methodologies.

[8] Exceptional behavior can be specified independently from other behavior in separate services and later combined.

[9] Services are specified by interaction patterns that define service access interfaces and protocols, as a significant part of a service description.

[10] The approach enables a direct mapping of a service model onto target architectures. Code generators exist that generate executable prototypes for complex distributed systems.

[11] The approach makes use of standard notations such as MSCs, structure diagrams and state transition diagrams. It provided extensions and own semantics to these notations.

[12] The methodology foresees the specification of QoS attributes, however it is not yet integrated in the approach.

[13] Due to the fact that services are described by formal models, discovery is possible in principle. However, dynamic systems are not scope of the mobilSoft project and therefore discovery mechanisms are not under consideration.

[14] The approach inherently supports time specifications (real time properties).
[15] The approach supports important methodological operations (refinement, splicing, composition, partial application), but lacks an overall integrated development process that is directly applicable.
[16] Functional dependencies are expressed using required/provided attributations and their interpretations [43], or through consistency/completeness checks as part of the methodology.
[17] The methodology covers the handling of exceptional cases by detection of conflicts, resulting partiality, and canonical completion.
[18] The AutoFocus execution model and the resp. infrastructure is used for runtime support. Because the approach targets the embedded domain, only functionality for implementation and deployment is used.

6. Discussion

In the previous section we have compared several service-oriented approaches according to our evaluation criteria. We have provided a structured overview of their properties and have given an interpretation. Analyzing the comparison of Table 1 yields commonalities of the approaches and their differences. We conclude that the focus on functionality is the underlying paradigm of all service-oriented approaches, putting functionality in the center of interest and providing means to specify functionality separately and combine it.

We also conclude that there are significant differences between the approaches. Not only do they target different development process phases, but they also have different goals motivated by specific problems which often are inherent to the application domains. For instance, managing interactions is a major concern in the domain of embedded distributed systems. For web service based systems, dynamic recovery and assembly of services is the main interest.

We can state that all service-oriented approaches provide benefits in terms of decoupling functionality from technical details and providing structured views onto system functionality. Thus, the approaches help to develop more robust, flexible and reusable systems.

7. Summary and Outlook

In this paper, we have given an overview of several approaches and methodologies around the notion of service, feature and function. First, we have described the usefulness of such concepts by looking at the different stages in the development process and their views on the system with different level of abstraction and detail. We have distinguished the stages requirements engineering, modeling and architecture design, implementation and integration, and deployment and run-time. Second, we have shown existing proven development approaches for Business Information Systems using Web Services technologies and communication middleware, and for telecommunication systems, where

the notion of feature is prevalent. Third, we have presented several service-oriented approaches of our group, namely the InServe, MEwaDis, mobilSoft, JANUS/FOCUS and Functional Architecture Modeling approaches.

In an effort to compare the different approaches we have identified and explained 15 distinct evaluation criteria. Each criterion and sub-criterion has an explanation and a discussion section. The discussion shows the importance of the respective aspect in a service-oriented approach and possible benefits and advantages. If there is a trade-off, we have shown both sides of the coin. We have compared 7 service-oriented development approaches using the evaluation criteria and have presented the results in a table with side remarks where necessary.

With this paper, we contribute to a common understanding of the terms service and service-oriented development and to general comparability of service-oriented development approaches from different application domains with different intents and purposes. This knowledge can lead to better interoperability and to the development of a comprehensive service notion, which is independent of any specific context.

Future work will include the comparison of additional service-oriented notions and approaches, and steps towards establishing a common service-oriented terminology and comprehensive, domain-independent service-oriented approach with mappings to the existing approaches. Furthermore, we will work on a formalization of a common subset of this approach and on tool support.

8. Bibliography

[1] D. S. Platt, K. Ballinger: *Introducing Microsoft .NET.* Microsoft Press, 2001.

[2] Aaron Walsh. *UDDI, SOAP, and WSDL: The Web Services Specification Reference Book.* Prentice Hall, 2002.

[3] Pamela Zave. *Feature-oriented description, formal methods, and DFC.* In Proceedings of the FIREworks Workshop on Language Constructs for Describing Features, pp 11–26. Springer-Verlag, 2001.

[4] Pamela Zave, *An experiment in feature engineering*. In: Annabelle McIver and Carroll Morgan, editors, Programming Methodology, pages 353-377, Springer-Verlag, New York, 2003.

[5] R. Orfali, D. Harkey, J. Edwards. *Instant CORBA*, Wiley, 1997.

[6] SUN Microsystems Inc. Java Platform, Enterprise Edition (Java EE, J2EE). Available at http://java.sun.com/javaee/. Version of 15-Nov-2006.

[7] M. Broy, N. Diernhofer, J. Grünbauer, M. Meisinger, M. Rappl, S. Rittmann, B. Schätz, M. Schoenmakers, B. Spanfelner. *Service-Oriented Development - Whitepaper*. Lehrstuhl für Software & Systems Engineering, Technische Universität München, 2006.

[8] InServe project homepage. Available at http://www4.in.tum.de/proj/inserve/. Version of 15-Nov-06.

[9] mobilSoft project's homepage. http://www.mobilsoft.info. Version of 01-Oct-2005.

[10] MEwaDis project homepage. Available at http://www4.in.tum.de/~mewadis/. Version of 15-Nov-06.
[11] M. Deubler. Dissertation. Technische Universität München, to appear, 2007.
[12] S. Rittmann: *Exploring Service-Oriented Software Development for Automotive Systems*. Diplomarbeit, Technische Universität München, 2004.
[13] B. Schätz, A. Pretschner, F. Huber, J. Philipps. *Model-based Development of Embedded Systems*. Technical Report TUMI-0402, TU München, 2002.
[14] P. Braun, M. v. d. Beeck, M. Rappl, C. Schröder. Automotive Software Development: A Model-Based Approach. *In-Vehicle Software*, SAE Technical Series, 2002.
[15] M. Broy. *Service-Oriented Systems Eingineering: Specification and Design of Services and Layered Architectures – The Janus Approach.* In: Engineering Theories of Software Intensive Systems, pp. 47-81. Springer, 2005.
[16] I. Krüger, R. Mathew, M. Meisinger: *Efficient Exploration of Service-Oriented Architectures using Aspects*. In Proceedings of the 28th International Conference on Software Engineering (ICSE), 2006.
[17] W3C: *Web Service Description Language 1.1.* Available at: http://www.w3.org/TR/wsdl. 12-Mar-2001 .
[18] C. Salzmann. *Modellbasierter Entwurf spontaner Komponentensysteme*. Dissertation, Technische Universität München, 2002.
[19] W3C: *Web Services Architecture*. Specification available at: www.w3.org/TR/ws-arch/. 11-Feb-2004.
[20] Werner Vogels. *Web Services Are Not Distributed Objects*. IEEE Internet Computing, vol. 07, no. 6, pp. 59-66, 2003.
[21] W3C: *Web Service Choreography Interface 1.0.* W3C Note, 8-Aug-2002. Available at http://www.w3.org/TR/wsci/
[22] Barry & Associates, Inc. *Web services and service-oriented architectures*. Website available at http://www.service-architecture.com/. Version of 15-Nov-06.
[23] OASIS: *UDDI Specification*. Available at: http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm (as of 25.11.2005)
[24] OASIS: *ebXML Homepage*. Available at: http://www.ebxml.org/ (as of 25.11.2005)
[25] OASIS. Web Services Business Process Execution Language (WS-BPEL), Version 2.0. Specification public draft available at http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.pdf, 23-Aug-2006.
[26] F. Leymann. *Web Services Flow Language (WSFL 1.0).* Technical report, IBM Corporation, 2001.
[27] Pamela Zave and Michael Jackson. New feature interactions in mobile and Multimedia telecommunication services. *Feature Interactions in Telecommunications and Software Systems VI*, pages 51–66, 2000. IOS Press, available at: http://www.research.att.com/~pamela/fiw6.pdf.
[28] DFC Homepage, available at: http://www.research.att.com/~pamela/dfc.html (as of 24.10.05)
[29] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *A System of Patterns. Pattern-Oriented Software Architecture*. Wiley, 1996.
[30] InServe Projektantrag. Accepted and funded by the Deutsche Forschungsgemeinschaft (DFG), 2004.
[31] I. Krüger. Distributed system design with message sequence charts. PhD Thesis, Technische Universität München, 2000.
[32] V. Ermagan, T. Huang, I. Krüger, M. Meisinger, M. Menarini, P. Moorthy. Towards Tool Support for Service-Oriented Development of Embedded Automotive Systems. In proceedings of the Dagstuhl Seminar on Modellbasierte Entwicklung eingebetteter Systeme (MBEES'07), to appear in LNCS, 2007.

[33] Technische Universität München and Validas AG. *AutoFocus project*. http://autofocus.informatik.tu-muenchen.de/index-e.html. Version of 15-Nov-06.

[34] M. Meisinger. *Service-Oriented Architecture Development of Reactive Systems. Dissertation*, Technische Universität München, To appear 2007.

[35] G. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison Wesley, 2003.

[36] M. Deubler, J. Grünbauer, A. Holzbach, G. Popp, G. Wimmel. *Kontextadaptivität in dienstbasierten Softwaresystemen*. Technischer Bericht TUM-I0511, TU München, 2005.

[37] M. Deubler, J. Grünbauer, G. Popp, G. Wimmel, C. 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.

[38] M. Deubler, J. Grünbauer, J. Jürjens, G. Wimmel. Sound Development of Secure Service-based Systems. In *International Conference on Service Oriented Computing (ICSOC)*, 2004.

[39] mobilSoft Projektantrag

[40] A. Fleischmann, J. Hartmann, C. Pfaller, M. Rappl, S. Rittmann, and D. Wild. *Concretization and Formalization of Requirements for Automotive Embedded Software Systems Development*. Accepted at the 10th Australian Workshop on Requirements Engineering (AWRE) 2005, Melbourne, Australia.

[41] S. Rittmann, A. Fleischmann, J. Hartmann, C. Pfaller, M. Rappl, and D. Wild. *Integrating Service Specifications on Different Levels of Abstraction*. Accepted at the IEEE International Workshop on Service-Oriented System Engineering (SOSE) 2005, Beijing, China.

[42] B. Schätz. *Building Components from Functions*. In: Electronic Notes in Theoretical Computer Science, Volume 160. Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005, 2005.

[43] B. Schätz. *Refactoring Functional Architectures*. TU München, 2005.

[44] B. Schätz, C. Salzmann. Service-Based Systems Engineering: Consistent Combination of Services. In *Proceedings of ICFEM 2003, Fifth International Conference on Formal Engineering Methods*. Springer LNCS 2885, 2003

[45] L. Kof, B. Schätz, I. Thaler, A. Wisspeintner. Service-based development of embedded systems. In *Net.Object Days Conference, OOSE Workshop*, Erfurt, Germany, 2004.

[46] M. Broy, and K. Stølen. *Specification and Development of Interactive Systems: FOCUS on Streams, Interfaces, and Refinement*. Springer, 2001.

[47] M. Broy. *Formal models for service-oriented interfaces and layered architectures*. Slides for the Marktoberdorf Summer School, 2004.