Modeling Crosscutting Services with UML Sequence Diagrams

Martin Deubler, Michael Meisinger, Sabine Rittmann

Technische Universität München
Boltzmannstr. 3
85748 München, Germany

{deubler, meisinge, rittmann}@in.tum.de

Ingolf Krüger

Department of Computer Science
University of California, San Diego
La Jolla, CA 92093-0114, USA

ikrueger@cs.ucsd.edu

Abstract. Current software systems increasingly consist of distributed interacting components. The use of web services and similar middleware technologies strongly fosters such architectures. The complexity resulting from a high degree of interaction between distributed components – that we face with web service orchestration for example – poses severe problems. A promising approach to handle this intricacy is service-oriented development; in particular with a domain-unspecific service notion based on interaction patterns. Here, a *service* is defined by the interplay of distributed system entities, which can be modeled using UML Sequence Diagrams. However, we often face functionality that affects or is spanned across the behavior of other services; a similar concept to aspects in Aspect-Oriented Programming. In the service-oriented world, such aspects form *crosscutting services*. In this paper we show how to model those; we introduce aspect-oriented modeling techniques for UML Sequence Diagrams and show their usefulness by means of a running example.

1 Introduction

Today's software systems get increasingly complex. Complexity is observed in all software application domains: In business information systems, technical and administrative systems, as well as in embedded systems such as in avionics, automotive and telecommunications. Often, major sources of the complexity are *interactions between system components*. Current software architectures increasingly use distributed components as for instance seen when using web services [11] and service-oriented architectures [7].

Traditionally, system design focuses on the components that structure the system physically or logically. Components are modeled and implemented in their entirety and mostly independent from each other; they are integrated in subsequent steps. In situations as we face them today with large distributed systems composed of a multitude of interacting components, significant portions of the components' functionality are determined by handling these interactions appropriately. Specifying separate components correctly and completely is a very difficult task. Additionally, when it comes to expressing system-wide concerns like certain Quality-of-Service properties such as end-to-end timing deadlines, component oriented models fall short.

More appropriate for the development of interaction intensive distributed systems are therefore approaches which put interaction modeling in the center of concern. Viewing systems entirely and explicitly modeling the interactions between the components that constitute the system addresses the before mentioned issues with arbitrary and exceptional interaction combinations and overarching system aspects. In interaction centric system specifications, the components are described by the interactions they have to provide; a black-box view that hides internal component complexity. Considering the different functions, features, or services that the system offers provides a straightforward structuring of such interaction models. For today's large multi-functional distributed systems, such approaches provide the necessary flexibility in decoupling the separate functions while still modeling systems in their entirety.

We strongly propose service-oriented software development approaches that place the different functions or services of a system in the center of interest – as for instance introduced by [14] or in [13]. We combine this with an interaction-centric development approach and specify the system services in terms of the interactions between the components involved [21]. We in particular use interaction-centric description techniques such as UML Sequence Diagrams or Message Sequence Charts [19], [22].

Problem Statement. When specifying a system in terms of services, it is often desirable to specify certain aspects of behavior contributing to or overlapping several existing services. We view such behavior again as services of the system, namely as *crosscutting services* or *aspects*. Examples for such crosscutting services are authentication, logging and synchronization. It is highly desirable to specify these crosscutting services separately in order to ensure a better comprehensibility, reusability, traceability and evolvability of the software models.

We achieve this by specifying each spread concern within one single unit, called aspect. These aspects – or crosscutting services – can then be modified independently from the rest of the system specification. These concepts are directly based on aspect-oriented programming and modeling techniques [18], [35].

Currently, UML sequence diagrams do not provide such aspect-oriented modeling techniques and are therefore not fully suited for systematic service-based software development. If a system needs to be specified precisely and without redundancy, more powerful notations and description techniques are required.

Contribution. In this paper, we address the mentioned problem by introducing an extension of UML Sequence Diagrams motivated by the ideas of aspect-orientation. We model a system based on interaction-based services using UML Sequence Diagrams. We extend the UML 2.0 Sequence Diagram Notation to enable the modeling of crosscutting behavior that is spread over the basic system interactions (services). Our approach is *independent of a specific domain*. We use a running example to explain and evaluate our notation. A more extensive version of this material can be found in [33].

In this paper, we initially assume a control-flow oriented, RPC-style (remote procedure call) communications paradigm for the execution of system services. This is a very common communication scenario that can be often observed for instance when using web services. We have shown in [19], [21] that our service-oriented approach also generalizes for asynchronous communication.

Outline. The remainder of the paper is structured as follows. In Section 2 we introduce our service notion and the sequence diagram based description techniques we use. Section 3 introduces our modeling approach and explains the extensions to the UML we propose. Section 4 discusses our approach in the context of related work. In Section 5, we present a conclusion and give an outlook on further steps.

2 Service-Oriented Development

In this section we briefly introduce our notion of service and service-oriented development. We specify services in terms of interaction patterns between system components and model those using UML Sequence Diagrams. Our UML extensions for specifying service aspects on top of existing system services are based thereon.

2.1 Services and Service Notion

We define a service as follows:

"A service is defined by the interaction among entities involved in establishing a particular functionality."

A *service* therefore is a *piece of behavior* or *functionality* which is provided by the collaborative inter-working of system entities. Hereby, an *entity* is an abstract, logical, structural part of the system. Depending on the level of detail it can stand for a component, module, package, or class. The *interaction* is described by interaction patterns that capture the message exchange between the system entities involved in establishing the service. As a consequence of the definition above, a service – and therefore functionality – can be spread across several entities or components. Note that in this paper we use the terms *entity* and *component* equivalently. Analogously, we speak of *functionality* or *behavior* when referring to a *service*.

Consider for instance the central locking system of a modern luxury car. It provides the service of unlocking the car remotely by pressing on the open button of the remote key. On doing so, the doors are unlocked, the alarm device and the anti-theft device are disabled, the exterior lights flash, the interior lights are turned on, the driver's seat is positioned, etc. As we can see, the remote unlocking service is provided by the collaborative work between the entities remote key, the door locks, the security devices, the exterior lights, the interior lights and the motor managing the driver's seat. They communicate with each other using messages.

In addition to capturing interactions between system entities, we also specify *local actions* of entities – for instance a computation of a certain result in reaction to the receipt of a message, to be sent to another entity. Note that a simple form of a service might not involve an interaction of multiple entities; services can be provided by just one entity. For instance, consider the service of adjusting the front seats individually by moving them forth and back.

Because our interaction-based service definition is founded only on the abstract interaction relationship between entities, we profit from the following advantages:

Our service notion can be used independently from a specific domain. A single methodology and supporting tool set can be applied in many different contexts.

Our service notion can be used throughout the overall development process – from requirements elicitation to implementation. Services are first class modeling elements that drive the entire process and that can be traced from requirements to implementation.

Our service notion goes beyond notions that define services by a callable list of procedures only. This is often seen when defining web services or network protocol stack service access points (SAPs). Our service notion instead enables elaborate behavioral specifications containing quality-of-service attributes.

2.2 Service Specification with UML Sequence Diagrams

In this section we show how services can be modeled by means of UML sequence diagrams. These and similar graphical notations and languages such as Message Sequence Charts and variants are well-suited ways of modeling interactions, by which our services are specified.

The Unified Modeling Language (UML) [42] has become the de-facto standard for modeling systems. The version 2.0 of the UML enhances the possibility of modeling complex and hierarchical interactions. It provides flexible and powerful constructs and operators to express conditions, parallel execution, repetition and hierarchy. We use UML Sequence Diagrams, to specify our services.

Figure 1 shows the simplified specification of a service of an automotive central locking system. The service is responsible for locking the trunk and all four doors. After pressing the central lock button (which is located inside the car), messages are sent to the trunk lock and the door locks, respectively. The figure shows the application of operators within the sequence diagram to express more complex interactions. In the example, locking the trunk happens in parallel (*par*) to locking the door. By design choice, all doors are locked in sequence. We could also have applied another parallel operator here. The service is established by the inter-working of the components CenterLockButton, TrunkLock and the four doors locks which communicate with each other by exchanging messages. The interaction between these system entities is captured in the interaction pattern that makes up the service.

We interpret the sequence diagram that is assigned to the service universally. This means, once the service is executed, the specified pattern of interactions must occur. In case of alternatives, the respective operator (*alt*) must be applied. Besides sequential and parallel composition of messages and alternatives, there are also loops, optional interactions and references to other interaction diagrams that can be expressed. More complex services can be composed by the use of interaction overview diagrams.

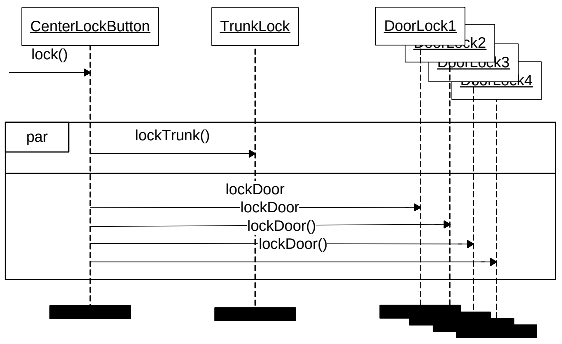


Figure 1: LockDoorsService

Note that with the UML 2.0 [42], sequence diagrams have converged much closer to the ITU standard of MSCs which have the advantage to be precisely defined; there is work existing that associates formal semantics to MSCs as well as distinct extensions that make MSCs even more suitable to service-based development approaches [19], [20], [22]. In this work, we focus on UML 2.0 Sequence Diagrams, because UML is more popular than MSCs in the context of modeling object-oriented systems, web services and synchronously communicating systems. Besides, there are many similarities and notational elements are directly transferable.

In this section we presented the notion of service we use as basis of our work. Methodological work around this notion of service can be found in [23], [33]. A formalization of services and service-oriented software architectures based on the mathematical model of streams can be found in [9].

3 Modeling Crosscutting Services with Aspects

Well structured software is usually divided into modules with certain responsibilities. This is in accordance with the separation of concerns principle simplifying reusability and a better maintenance of design and implementation. However, some behavior crosscuts these so-called primary modules as it affects several modular entities. In object-oriented programming, the units of modularity for example are classes; a crosscutting concern is spanned across several classes. If the implementation of such a concern is scattered and tangled up with the core functionality, it is difficult to reason about, implement and change. This problem is called the tyranny of the dominant decomposition [39].

Aspect-oriented programming (AOP, cf. [1], [8]) is an implementation level technology that allows to isolate pieces of behavior into single units – called *aspects*; it allows to specify at which locations in the code the aspects should later be inserted.

This ensures the encapsulation of cross-cutting behavior, such as logging or synchronization and therefore results in a better comprehensibility, reusability, traceability and evolvability of the code.

Aspect-oriented modeling (AOM) is a consequence of aspect-oriented programming; it raises the ideas of separation of concerns to the level of software models. To date, most of the work on aspect-orientation has concentrated on the implementation level. How aspects can be modeled appropriately is still not investigated sufficiently. This is particularly true in the area of service-oriented modeling where we face the before mentioned problems. However, in order be able to create more powerful, more elaborate system models, it is inevitable to also model crosscutting services. Therefore it is necessary to have a way to apply aspect-orientation in models.

In the following we explain what aspects are in the service-oriented world; we list the goals and principles that are relevant when modeling aspects. In Section 3.3 we introduce our approach to model crosscutting services using an extended Sequence Diagram notation.

3.1 Crosscutting Services

The notion of *aspect* has emerged during the last years (cf. [3], [4], [5]). E.g. in [27] an informal definition for the term aspect is given. The author describes an aspect as a “crosscutting concern”. Here, a concern is a “property of interest to a stakeholder” and crosscutting means “intertwining, interdependent, interacting, [or] overlapping”.

In component-based software development approaches, the modular entities are components (or classes, packages, etc. depending on the level of examination). A crosscutting concern therefore is overlapping or affecting several components. In our service-oriented point of view, services are by definition overlapping several components. Therefore, each service would be an aspect, which renders the above definition of crosscutting concerns impractical. Instead, in the service-oriented world, the basic building blocks are services. Consequently, an aspect – or crosscutting service – is a service that is spanned across or that influences the behavior of other services.

It shows that our definition of crosscutting concerns in the context of service-oriented development follows a similar idea as aspects in a component-oriented approach. We allow specification of modular pieces of behavior and separate crosscutting concerns in aspects. However, our approach retains all before mentioned advantages and benefits from interaction based modeling.

In the following we distinguish between *crosscutting* and *crosscut* behavior. Crosscutting behavior is the functionality that is spread over several services. It affects behavior of existing services which in turn are the *crosscut* functionality.

3.2 Principles and Goals for Modeling Crosscutting Services

Introducing a notation for modeling of crosscutting services should comply with the following principles and goals:

Model crosscutting services like basic services with additional characteristics: A crosscutting service is a service that affects the behavior of other services. Therefore, it is suggestive to model crosscutting behavior similar to basic services: with Sequence Diagrams. However, we have to take additional care of the special characteristics that make up the nature of service aspects.

Cleanly modularize crosscutting services: In order to enable reuse and a better maintenance of design (and later: implementation) we have to cleanly modularize the crosscutting service. Each aspect needs to be located within one model.

Leave crosscut service untouched: An aspect might affect several services. However, if a crosscut service should be reused in another system or configuration without the crosscutting service, it will not be influenced by the crosscutting service anymore. Therefore, the specification of the crosscut service must remain independent and unchanged by the aspect. The dependency is unidirectional: the crosscutting service depends on the execution context of the crosscut service.

Attend to clear illustration of crosscutting relationships: Crosscutting relationships can be very complex. For example, if several services are affected by several other services or aspects that are crosscut by aspects in turn. A modeling technique to capture/specify aspects must illustrate these complex dependencies concisely.

Additionally, a good modeling approach should provide both a coarse-grained (more abstract) and a fine-grained (more concrete) view on crosscutting relationships. In the next section we will introduce a modeling approach that is in accordance with the listed principles. However, we will not show how aspects can be incorporated in *structural* diagrams such as UML Class Diagrams; the focus of our work lies on the *behavioral* part.

3.3 Modeling Crosscutting Services with Sequence Diagrams

Crosscutting services differ from basic services as explained before. We need to specify additional characteristics when modeling aspects. Plain sequence diagrams do not provide enough flexibility and expressiveness. Consequently, we have to introduce additional modeling elements that provide the required expressiveness. In the following we will introduce our modeling elements step by step by means of an example.

Modeling Join Points

In contrast to basic services, crosscutting services model *when* the crosscutting behavior takes *place in reference to the behavior of affected services*. We have to specify the points in the system execution where an aspect starts, affects and ends.

In order to define the places where two concerns crosscut one another, we introduce elementary ***join points.*** We adapt AspectJ [2] nomenclature here. Nonetheless, the concept used permits the use of various AOP flavors. We do not show how our modeling concepts can be translated into AspectJ as they are independent of a particular programming language. Join points mark well-defined, single points in the execution flow at which two concern models are (inter)connected with each other. Join points correspond to messages and local activities of sequence diagrams.

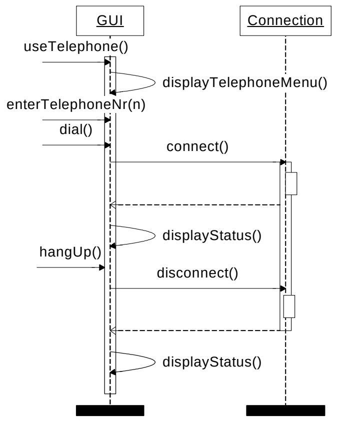
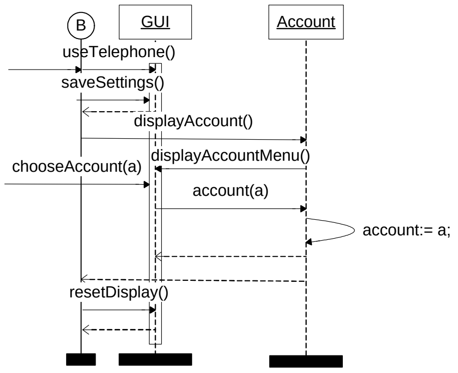


Figure 2: UseTelephoneService

In particular, we specify when a crosscutting concern should be executed. We face the following possibilities: Before, After, Around or Instead of a certain action. Actions can be local activities or message send or receive events. We introduce graphical elements to specify when an aspect is invoked – namely *before, after, wrapping* and *overriding* join points. In the following we will explain them with the aid of examples: Figure 2 shows a simple telephone service which can be found for instance in a modern luxury car. When the telephone menu is selected via the car's user interface (MMI), the user can enter a phone number, and connect to this number. The system establishes the connection and displays the call status until the user hangs up. The system disconnects the call and updates the status display.

Now assume that the telephone costs should be charged to individual users. This can be done by enabling an *AccountService*. An account has to be chosen before the actual TelephoneMenu can be used. How can this be realized? One possibility would be to insert the new behavior – the choice of a specific account prior to the use of the telephone – directly in the *UseTelephoneService*. However, it is better to modularize the account service in a separate module as it can be enabled and disabled.

Before Join Points. We have to model the point in the execution flow where the behavior is affected. For our example, we choose to insert the new behavior when the telephone service is called, but *before* it is actually performed. Figure 3 shows the introduction of the before join point symbol. The message *useTelephone()* is divided by an axis starting from a before join point (a circled "*B*"). The semantics is that the message *useTelephone()* is not delivered to the axis *GUI.* Instead, the crosscutting behavior is performed: The current display settings are saved (*saveSettings()*) and an account menu is shown (*displayAccountMenu()*). After an account has been chosen (*chooseAccount()*), the display is reset to the saved settings (*resetDisplay()*). The very last, unlabeled arrow indicates that the control flow is given back to the before join point. This means that the afore interrupted message *useTelephone()* is now actually delivered to the *GUI*. That is the point in the execution where the crosscutting behavior ends and the crosscut behavior is continued.



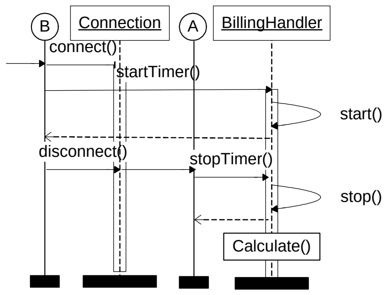


Figure 3: AccountService

Figure 4: BillingService

After and Wrapping Join Points. The *BillingService* (see Figure 4) shows the modeling elements for after and wrapping join points. This service has two parts: (1) When a connection is requested (*connect()*), a timer is started, and (2) When the connection is closed (*disconnect()*), the timer is stopped. Then, some calculation is performed.

In the first part we again make use of a *before* join point. In the second part we introduce an after join point having the following semantics: After the message *disconnect()* is sent to Connection, the message *stopTimer()* is sent. When the timer is stopped, the control flow is returned to the after join point. The behavior being crosscut continues its execution.

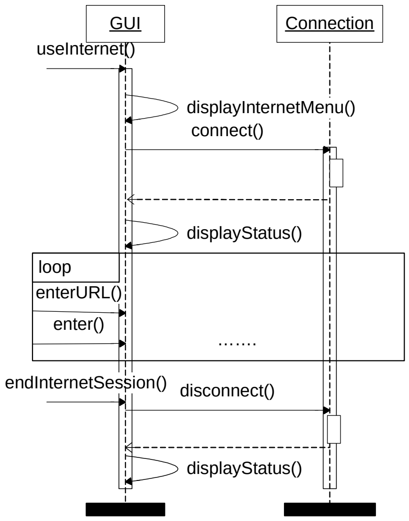
In the above example we introduced not only before and after join points. In fact we specified a wrapping aspect which has defined start and end points, respectively. The issue of modeling overriding join points is currently being investigated. Affecting local activities can be obtained similarly to messages.

Combination of Join Points – Point Cuts

Of course, we also could have modeled the crosscutting behavior as part of the *UseTelephone* service. However, we cleanly isolated the aspect in a separate model. The advantage is evident if we also introduce an internet service (cf. Figure 5).

Applying the account service also to the internet service is now simple by adding a combination of join points (point cuts in AspectJ nomenclature) to the aspect specification. Figure 6 shows how the account service is specified so that it is applicable to either the telephone or the internet service. The *alt*-box defines a logical combination of join points – to be more precise: the logical "OR" between two *before* join points. Either before the message *useTelephone()* or before the message *useInternet()* is sent to the *GUI*, the behavior is interrupted.

Another possibility for the specification of point cuts (combinations of join points) is the use of *parallel-boxes*, etc. The concept of point cuts allows us to model the logical combination of several points in order to specify more complex points in the program execution. Point cuts pick out certain join points in the program flow and values at those points. Investigating this in more detail is one of our next goals.



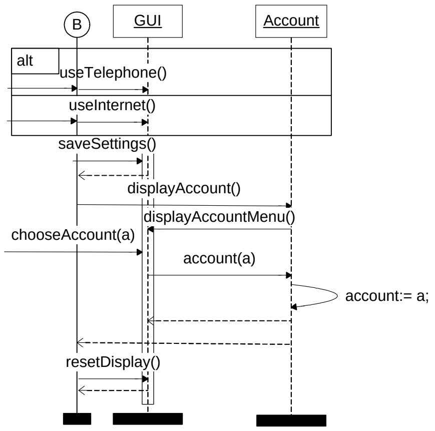


Figure 5: UseInternetService **Figure 6: AccountService w. Join Point**

Modeling of Execution Context

To provide aspect services with higher flexibility and expressiveness, we expose the execution context of the affected services in point cuts. The aspect thus can make use of it. In the sequence diagrams, we add OCL-style notes to the arrows. Figure 7 shows a report service that reports how much an account has to be charged for. For this purpose, the *sum* (which are the costs of the internet or telephone session) and the *account* (which is to be charged for) are available to the reporting service. They can be seen as parameters being provided to the report service.

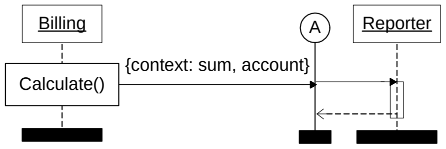


Figure 7: Report Service

Name-Based and Property-Based Specification

In the examples above, we explicitly specified concrete message and activity names when determining the crosscut behavior. We call this *name-based crosscutting.* Sometimes, crosscutting behavior affects many other services. In this case it would mean much effort to specify all places where the aspect makes an appearance. A more powerful way to specify the location of join points within sequence diagrams is *property-based crosscutting.* For example we can use wildcards to specify a group of messages, activities or even axes. Instead of using the *alternative-box* in Figure 6, we could just write *use*()*. A property-based crosscutting service can be seen as a template which is instantiated by each message fulfilling the property. Another possibility is to use parameterized messages. Then a set of possible messages substituting the parameters would be specified.

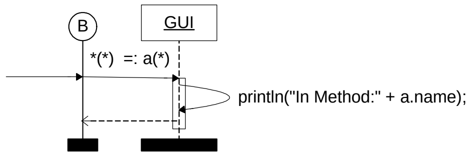


Figure 8: Trace Service

This mechanism is especially useful if some crosscutting behavior affects many services at different points. Let us assume that we want to trace the execution of a program for example. To that end, we define an aspect that prints the name of each method call. In Figure 8 the method name of each method (having an arbitrary number of arguments) called to *GUI* is printed *before* the method is actually executed.

4 Discussion and Related Work

In this section, we put the service notion and our introduced aspect-oriented description techniques that we have presented in the preceding sections into perspective; in particular, we discuss our approach in relation to others known from the literature.

As noted in [40], the term *service* is used in a variety of meanings, and on various levels of abstraction in the Software Engineering community. The notion and model of service we use in this document captures the interplay of multiple components collaborating to achieve a particular function or feature of the system under consideration. This encompasses the various “traditional” notions of service used in the telecommunications [41] and business information systems domains, but also the emerging uses of the term service in the context of “web services” [11], and service-oriented architectures (SOAs) [7].

In the telecommunications domain, the notion of feature is well established and researched – as pointed out in [41]. Features can be defined as “reusable, self-contained services” [30]; they encapsulate individual pieces of functionality of limited scope, typically used to structure the interfaces or internals of components. Feature-oriented software design and development [31] makes use of features as principal modeling elements. According to [17] features are units of “observable behavior”, and “requirements modules” serving as “units of incrementation as systems evolve.”

On the other hand, web services [11], [34] at first glance define simple call/reply relationships between the consumer and the provider of the web service. At closer inspection, however, it becomes apparent that web services and their supporting architectures are more loosely-coupled than traditional layered architectures. In particular, web-services typically emerge from the interplay of multiple components – a call upon one web service, in general, results in calls upon multiple other (web) services provided by other components or applications. A web service thus acts as an orchestrator for the collaboration of the components implementing the functionality “behind” the service. This view of services as orchestrators of collaboration is becoming increasingly popular [28], [14]; it transcends the realm of web services – where it is prominently recognized, for instance, by the business process execution language for web services (BPEL4WS) and takes root also in the domain of complex embedded systems as found in the automotive [6] and avionics domains [32].

We have demonstrated the use of UML Sequence Diagrams as graphical description techniques for services. Because Sequence Diagrams capture interaction behaviors that cut across multiple components, we view services and their graphical representations in the form of UML Sequence Diagrams as modeling aspects in analogy to the implementation aspects captured by aspect-oriented programming languages such as AspectJ [18], [2]. In fact, we have shown in [21] that services can be translated immediately into corresponding AspectJ programs; the weaving mechanism of AspectJ can then be exploited to integrate the services defined as Sequence Diagrams into a correct set of component implementation.

Unfortunately, the modeling of aspects is not supported by MSCs or UML sequence diagrams. Although work on including aspect-oriented concepts in the UML has recently been published (cf. [8], [9], [10], [35], [12] and [36]), no elaborate notational elements exist in order to model aspect-orientation adequately. Furthermore, the cited work mainly introduces concepts for class diagrams. Therefore, we give some first ideas on how the modeling of cross-cutting services by means of interaction diagrams can be done.

Recent work has been published that relates aspect-orientation to requirements engineering and design phases, and modeling, cf. [1], [26], [43], [36], [37]. Our service notion can be seen as precisely formulated requirements specified as sequence diagrams. We put our focus in particular on the architecture definition and design phase. The work in [1] mainly sees aspects as crosscutting non-functional requirements or in particular quality attributes [26], while we show how to model crosscutting functional behavior (services) in a similar way to basic services, using cautiously extended sequence diagrams. Similar to [43], we see aspects as interaction patterns. However, we focus on service executions with RPC-style communication semantics which are often used for web service combinations; we also introduce explicit notations to model before, after, around and instead join points within sequence diagrams. In this way we differ from [36] that uses standard UML concepts to represent basic and crosscutting behavior, which we consider too limiting and less intuitive for our purposes. The work in [37] focuses on structural concerns in aspect-oriented design with UML and aspect information interchange using XML, while we focus on behavior models and put the notion of service in the center of concern.

Our approach is related to Model-Driven Architecture (MDA) [25], Model-Integrated Computing [38], aspect-oriented modeling (AOM) [15] and architecture-centric software development (ACD) [42]; similar to MDA and ACD we also separate the software architecture into abstract and concrete models, as for instance shown in [21]. In contrast to the cited model-driven development approaches, however, we consider services and their defining interaction patterns as first-class modeling elements of all our models throughout the different development phases.

In Section 3.2 we mentioned the importance of having both a coarse-grained (more abstract) and a fine-grained (more concrete) view on crosscutting relationships. In this paper we only showed the detailed view of the intertwined relationships. The other case can for instance be achieved by introducing additional stereotypes for UML Use Case Diagrams (cf. [33] for more information).

5 Summary and Outlook

In this paper we have shown the significance of an interaction-oriented development approach to address the always increasing complexity of current software systems. Because significant sources of software complexity stem from the interactions among interacting components, an interaction-based model targets the problem at its source. We explained service-oriented development as an approach to specify and develop systems in terms of services as first class modeling elements. Services are defined in terms of interaction patterns between components that participate in the service. Services as such realize crosscutting behavior that spans multiple components.

In analogy to the introduction of aspect-oriented technologies for component or object-oriented software development approaches, we showed its usefulness also for service-oriented development. Retaining all advantages of system specifications using services, we provide means to separate specific pieces of behavior that affect multiple services into aspects. Aspects again are services: crosscutting services. We introduced a notation as an extension to UML 2.0 Sequence Diagrams to model services and services that cross-cut services using aspect-oriented techniques.

We have explained our notation using a running example that is representative for many different domains, including telecommunications, automotive and web service based applications. It is representative for similar situations such as the composition of systems out of interacting web services. We showed that our approach is well suitable to model the traditional RPC-style interactions that occur when composing a system out of a number of separate services.

In the future, further investigations have to be done in how to specify non-functional crosscutting concerns, such as performance or timing constraints. Also, we want to investigate how to identify and resolve contradictory aspect specifications. To prove the value and efficacy of our approach, we plan to apply it to case studies of significant size within the automotive domain as well as in the web services area.

In our work, we concentrate on the *behavioral* part of aspects. The *structural* part – for instance how aspects can be modeled in class diagrams – is not focus of our work. However, the relation between behavioral and structural modeling of aspects has to be investigated in the future, too.

Acknowlegements

Our work was partially supported by the Deutsche Forschungsgemeinschaft (DFG) within the project *InServe* and by the Bavarian high-tech funding program (High-

Tech Offensive) within the project *MEWADIS*. Further funds were provided by the UC Discovery Grant and the Industry-University Cooperative Research Program, as well as by the California Institute for Telecommunications and Information Technology (Calit2). We are grateful to the anonymous reviewers for insightful comments.

References

[1] J. Araujo, A. Moreira, I. Brito, A. Rashid: Aspect-oriented requirements with UML. In Proceedings of the Workshop on Aspect-oriented Modeling with UML, UML 2002, Dresden, Germany, October 2002.
[2] AspectJ Team: The AspectJ Programming Guide. Available at http://eclipse.org/aspectj/.
[3] Aspect-Oriented Software Development. Proceedings of the 1st international conference on Aspect-oriented software development. ACM Press, 2002.
[4] Aspect-Oriented Software Development. Proceedings of the 2nd international conference on Aspect-oriented software development. ACM Press, 2003.
[5] Aspect-Oriented Software Development. Proceedings of the 3rd international conference on Aspect-oriented software development. ACM Press, 2004.
[6] Automotive Open System Architecture, www.autosar.org
[7] L. Baresi, R. Heckel, S. Thone, D. Varro: Modeling and validation of service-oriented architectures: Application vs. style. In Proc. of ESEC/FSE, 2003.
[8] M. Basch, A. Sanchez: Incorporating aspects into the UML. In Proceedings of the International Conference on Aspect-Oriented Software Development, March 2003.
[9] M. Broy, I. Krüger, M. Meisinger: Services and service-oriented software architectures – methodological foundations. To appear.
[10] S. Clarke , R.J.Walker. Composition patterns: An approach to designing reusable aspects. In Proceedings of the 23rd International Conference on Software Engineering, pp. 5–14, May 2001.
[11] A. Colin: Why web services? The Web Services Industry Portal, February 2002. Available at http://www.webservices.org/index.php/article/articleprint/75/-1/61/.
[12] C.A. Constantinides. A case study on making the transition from functional to fine-grained decomposition. In Proc. of ECOOP 2003 Workshop on Analysis of Aspect-Oriented Software (AAOS 03), July 2003.
[13] M. Deubler, J. Grünbauer, G. Popp, G. Wimmel, C. Salzmann. Tool Supported Development of Service Based Systems. In 11th Asia-Pacific Software Engineering Conference (APSEC 2004), IEEE Computer Society, Korea, 2004.
[14] E. Evans: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, 2003.
[15] G. Georg, R. France, and I. Ray: Composing Aspect Models. The 4th AOSD Modeling With UML Workshop, 2003.
[16] C. Ghezzi, M. Jazayeri, R. France. Fundamentals of Software Engineering. Prentice Hall, 1991.
[17] P. Gibson, D. Méry: Formal Modelling of Services for Getting a Better Understanding of the Feature Interaction Problem. In Bjorner, Broy, Zamulin (eds): Perspectives of System Informatics, Lecture Notes in Computer Science. Volume 1755, Springer, 2000.
[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold: An overview of AspectJ. In Proceedings of the 15th European Conference on Object-Oriented Programming ECOOP 2001, LNCS vol. 2072, pp. 327–353, Springer, June 2001.
[19] I. Krüger. Specifying services with UML and UML-RT. In Electronic Notes in Theoretical Computer Science, volume 65 (7). Elsevier Science B. V., 2002.

[20] I. Krüger: Service specification with MSCs and roles. In Proceedings of IASTED International Conference on Software Engineering, Innsbruck, 2004.
[21] I. Krüger, R. Mathew: Systematic development and exploration of service-oriented software architectures. In Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA 2004), 2004.
[22] I. Krüger: Towards precise service specification with UML and UML-RT. In Proceedings of the Workshop at UML, Critical Systems Development with UML (CSDUML), 2002.
[23] R. Mathew: Systematic definition, implementation and evaluation of service-oriented software architectures. Master Thesis at University of San Diego, California, 2004.
[24] Message Sequence Chart (MSC 96), ITU-T. Recommendation Z.120. ITU-T, 1996.
[25] Model Driven Architecture. Object Management Group. Available at http://www.omg.org/mda/, 2003.
[26] A. Moreira, J. Araujo, and I. Brito: Crosscutting Quality Attributes for Requirements Engineering. Software Engineering and Knowledge Engineering Conference (SEKE), 2002.
[27] B. Nuseibeh: Crosscutting Requirements. AOSD 2004, The Open University, UK, 2004.
[28] C. Peltz: Web Services Orchestration and Choreography. IEEE Computer 36(10): pp. 46-52, 2003.
[29] D.S. Platt, K. Ballinger: Introducing Microsoft .NET. Microsoft Press, 2001.
[30] C. Prehofer: Plug-and-Play Composition of Features and Feature Interactions with Statechart Diagrams. In Proc. of the Seventh International Workshop on Feature Interactions in Telecommunications and Software Systems, Ottawa, 2003.
[31] C. Prehofer: Feature Oriented Programming: A fresh look at objects, In Proceedings of ECOOP 1997, Springer LNCS 1241, 1997.
[32] Realtime CORBA Joint Revised Submission, Object Management Group, OMG Document orbos/99-02-12 ed., March 1999.
[33] S. Rittmann: Exploring Service-Oriented Software Development for Automotive Systems. Diplomarbeit, Technische Universität München, 2004.
[34] J. Snell, D. Tidwell, P. Kulchenko: Programming Web Services with SOAP. O'Reilly, 2002.
[35] G. Sousa, S. Soares, P. Borba, J. Castro: Separation of crosscutting concerns from requirements to design: Adapting an use case driven approach. In Proc. of Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design. Workshop at AOSD 2004, March 2004.
[36] D. Stein, S. Hanenberg, R. Unland: Designing aspect-oriented crosscutting in UML. In Proceedings of Aspect-Oriented Modeling with UML. As part of the 1st International Conference on Aspect-Oriented Software Development, April 2002.
[37] J. Suzuki, and Y. Yamamoto: Extending UML with Aspects: Aspect Support in the Design Phase. AOP Workshop at ECOOP'99, Lisbon, Portugal, 1999.
[38] J. Sztipanovits, and G. Karsai: Model-Integrated Computing. IEEE Computer, Apr. 1997, pp. 110-112.
[39] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton: N degrees of separation: Multi-dimensional separation of concerns. In Proceedings of the 21st International Conference on Software Engineering, May 1999.
[40] D. Trowbridge, U. Roxburgh, G. Hohpe, D. Manolescu, E.G. Nadhan: Integration Patterns. Patterns & Practices. Available at www.microsoft.com, 2004.
[41] K. J. Turner: Relating Services and Features in the Intelligent Network. In Proc. of the 4th International Conference on Telecommunications, pp. 235-243, Zagreb, June 1997
[42] UML 2.0. Object Management Group. Available at http://www.omg.org/uml.
[43] J. Whittle, and J. Araujo: Scenario Modeling with Aspects. IEE Proceedings - Software, Special Issue on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, August 2004.