Efficient Exploration of Service-Oriented Architectures using Aspects

Ingolf H. Krüger, Reena Mathew
Department of Computer Science
University of California, San Diego
La Jolla, CA 92093-0114, USA
{ikrueger,rmathew}@cs.ucsd.edu

Michael Meisinger
Institut für Informatik
Technische Universität München
Boltzmannstr. 3, 85748 Garching, Germany
meisinge@in.tum.de

ABSTRACT

An important step in the development of large-scale distributed, reactive systems is the design of *architectures* that effectively support the systems' purposes. Early *prototypes* help to decide upon the most effective architecture for a given situation. Questions to answer include the boundaries of components, communication topologies and of replication. It is desirable to *evaluate and compare* architectures for functionality and quality attributes before implementing or changing the whole system. Often, the effort required is prohibitive. In this paper we present an approach to *efficiently* create prototypes for service-oriented architectures using aspect-oriented programming techniques. We explain a procedure for transforming interaction based software specifications into AspectJ programs. We show how to map the same set of interaction scenarios to different candidate architectures. This significantly reduces the effort required to explore architectural alternatives. We explain and evaluate our approach using the Center TRACON Automation System as a running example.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Algorithms, Design, Measurement, Performance

Keywords

Services, Service-Oriented Development, Distributed Reactive Systems, Roles, Components, Software Architecture Exploration, Architecture Comparison, Aspect-Oriented Programming, Aspects, AspectJ

ICSE'06, May 20–28, 2006, Shanghai, China.

1. INTRODUCTION

Designing complex distributed systems is a difficult task. Selecting effective architectures is crucial for the success of these systems. The exploration of different architectural alternatives, however, often falls victim to time pressure and lack of resources: it typically involves writing large parts of the code for each alternative upfront to support the evaluation – this binds people, time and financial resources.

1.1 Problem Definition

Important questions that need to be addressed when designing architectures include, for instance, how to design the objects/components and their interfaces, how to connect and distribute them i.e. how to design the communication topology, and how to replicate components for most efficient operation on a given middleware. Answering these questions requires consideration and exploration of different alternative architecture candidates. Building prototypes and running simulations complements and provides input for architecture evaluation techniques based on reviews and estimation [2]. However, building multiple prototypes to systematically explore different architectural alternatives is costly.

One way to address this problem is to separate an overall software architecture into logical models (sometimes called *domain models* [4]) and implementation models; approaches advocating this separation are architecture-centric software development [31] and model-driven architecture [22].

A clear separation into logical and implementation models is often difficult to achieve – especially in situations where requirements suggest a tight coupling between the two types of models. This is often the case when requirements include specific performance and other Quality-of-Service properties. In such cases, exploring multiple architectural alternatives is often not an option, because writing "throw-away" prototypes is too costly: large parts of the deployment infrastructure have to be written and re-written for each prototype. Furthermore, the mapping between logical and implementation model is generally non-trivial and there are likely different mappings that need to be considered for different alternatives. Supporting multiple such explorative hand-crafted mappings can quickly become costly, too.

A core source of complexity is that the scenarios supported by the system typically involve a multitude of collaborating entities partaking in complex interactions. These interactions are part of both the logical and implementation models. A well-defined mapping needs to exist between the interactions in both models. In case there are different im-

plementation alternatives, this leads to re-implementing the same interplay over and over again.

Our goal is to provide a solution where a collaboration specification can be reused unchanged across all implementation models (or target-architectures) to be evaluated. The major step toward achieving this goal is to decouple the "features" or "services" a system provides from the architecture on which it is deployed.

1.2 Service-Oriented Specifications

In this paper, we propose an approach to architecture evaluation and exploration that establishes a clean separation between the services provided by the system under consideration, and the architecture – comprised of components and their relationships – implementing the services.

We use the notion of *service* to decouple abstract behavior from implementation architectures supporting it. The term "service" is used in multiple different meanings and on multiple different levels of abstraction throughout the Software Engineering community [30]. *Web Services* [29] currently receive a lot of attention from both academia and industry. Figure 1 shows a typical "layout" of applications composed of a set of (web) 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 facade to the underlying domain objects – in effect offering an interface that shields the domain objects from client software. For a web-service-based application the service layer consists of the functions this application exposes over the Internet following standards such as WSDL (for describing location, function name, parameter names and types), SOAP (for message encoding), and HTTP (for message transport). The domain layer consists of "plain old objects" representing the data and logic of the underlying application. Typically, services in this sense coordinate workflows among the domain objects; they may also call, and thus depend on, other services. Some of the services, say *Service 1* and *Service 2* in our example, may reside on the same physical machine, whereas others, such as *Service n* may be accessible remotely via the Internet.

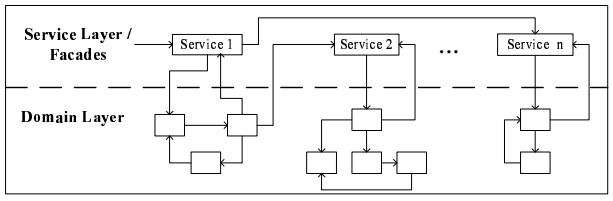


Figure 1: Service-Oriented Architectures

The layout shown in Fig. 1 is prototypical not only of the situation we find for applications structured in terms of web services, but also for other domains where complex, often distributed applications are expected to offer externally accessible interfaces. Indeed, service-oriented approaches to system development, leading to similar application structures, are prominent in the telecommunications domain [34] and are emerging in the automotive domain [1]. Abstracting from the domain-specific details we observe that services often encapsulate the coordination of sets of domain objects to implement "use cases".

We view services as specializations of use cases to specify interaction scenarios; services "orchestrate" the interaction among certain entities of the system under consideration to achieve a certain goal [4]. In contrast to use cases, which describe functionality typically in prose and on a coarse level of detail, we define a service via the interaction pattern among a set of collaborators required to deliver the functionality. Services are partial interaction specifications.

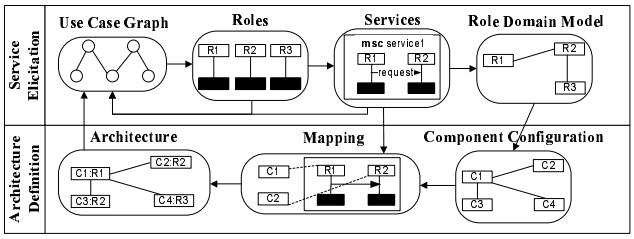


Figure 2: Service-Oriented Development Process

We employ a two-phase, iterative development process as shown in Fig. 2. Phase (1), *Service Elicitation*, consists of defining the set of services of interest – we call this set the service repository. Phase (2), *Architecture Definition*, consists of mapping the services to component configurations to define deployments of the architecture.

In phase (1) we identify the relevant use cases and their relationships in the form of a use case graph. This gives us a relatively large scale scenario-based view on the system. From the use cases, we derive sets of *roles* and *services* as interaction patterns among roles. Using roles decouples from interaction details, because roles abstract from components or objects. Roles describe the contribution of an entity to a particular service independently of what concrete implementation component will deliver this contribution. An object or component of the implementation typically will play multiple roles at the same time. The relationships between the roles, including aggregations and multiplicities, develop into the *role domain model*.

In phase (2) the role domain model is refined into a *component configuration*, onto which the set of services is *mapped* to yield an *architectural configuration*. These architectural configurations can be readily implemented and evaluated as target architectures for the system under consideration.

The process is iterative both within the two phases, and across: Role and service elicitation feeds back into the definition of the use case graph; architectures can be refined and refactored to yield new architectural configurations, which may lead to further refinement of the use cases.

1.3 Architecture Exploration and Aspects

Our service notion is based on interaction patterns that span across several entities implementing them; services emerge as independent system-wide aspects of both logical and concrete models. This observation motivates our use of aspect-oriented programming to capture services as aspects and to use weaving techniques [11] to establish the mapping between one abstract model and multiple concrete models. Aspect-orientation propagates the separation of cross-cutting concerns into aspects, which can be efficiently woven together to form the executable software. Note, that we use the term "cross-cutting" primarily in the sense of func-

tionality that involves the coordinated interplay of multiple components.

Exploiting this ability to efficiently generate multiple executable candidate architectures is critical to render architecture evaluation practical. AspectJ [10] provides the infrastructure we need to translate services into aspects. In Sect. 3 we explain this mapping in detail and show how the combination of services and aspects helps us to solve the problems explained above.

1.4 Contributions and Outline

As main contribution, this work presents a systematic approach to rapidly implementing software architecture exploration prototypes using an aspect-oriented programming language. We give a translation procedure from service models – scenario-based interaction specifications – to AspectJ implementations. We show how this translation can easily be automated and how the prototypes can be used to explore different architecture alternatives. A major significance of our approach is that it disentangles the specification of functionality (the services) from the infrastructure on which they are implemented; we can rapidly build and evaluate prototypes for different target architectures without having to rewrite large portions of the code. This results in reduced effort and time for performing an architecture evaluation.

In Sect. 2, we introduce the Center Tracon Automation System (CTAS) as our running example and show how it is modeled in terms of services. In Sect. 3, we explain how to translate the architecture definition to aspects for implementing the system. We also show how we make use of tools to support our approach. In Sect. 4, we report on experiences with evaluating various architectures for CTAS applying our approach; we also provide a brief evaluation. In Sect. 5 we show related work; we discuss our approach in Sect. 6. Sect. 7 contains conclusions and an outlook.

2. SERVICE-ORIENTED MODEL OF CTAS

To demonstrate our approach, we use the Center TRACON Automation System (CTAS), a case study from the air-traffic control domain, as an example of a large-scale distributed system [25]. CTAS is a set of tools and processes designed to help air traffic controllers manage the increasingly complex air traffic flows at large airports. An important part of this system is the distribution of weather updates to interested clients; this is the part we concentrate on in our case study.

In essence, this example implements a distributed version of the Observer pattern (cf. [6]) and its underlying infrastructure. It can be seen as a representative for a class of similar interaction intensive systems, including (a) business information systems with distributed components communicating via web services, and (b) database systems implementing distributed transactions by two-phase commit protocols. The time required to execute a service, or to commit a transaction to the database is one of the critical properties in the system. It is significantly determined by the communication infrastructure and component architecture of the system. This makes this parameter a natural target for architecture exploration.

As given in the requirements [25], the main component of the CTAS weather update system is the communications manager CM; other processes, including route analysis (RA), and the plan-view GUI (PGUI), are clients to CM. Clients are distinguished as *aware* or *unaware* depending on whether they participate in the weather update process. The CTAS requirements [25] explain how the clients initialize with CM, and how CM subsequently relays the latest weather information to all aware clients. For this paper, we design and implement various architectures and communication setups for the weather update functionality of the CTAS system as a refinement of the structure given above. CM continuously checks if a new weather report is available. If so, the CM sends a message to all aware clients. Each client responds, indicating whether it can process the weather update successfully. If all clients indicate success, the CM asks all the clients to use the latest weather information. If at least one of the clients indicates failure, the CM informs all clients to use the old weather information. This is important to ensure the consistency of this critical information in all parts of the system. The time it takes for an update cycle to complete is an important performance property of CTAS.

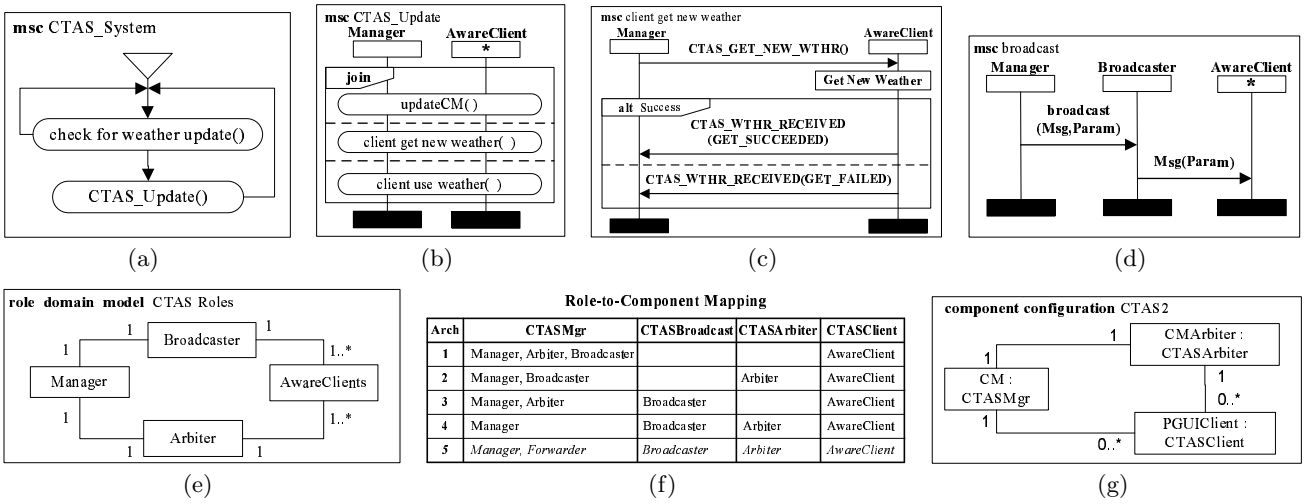
In the following we show how to define, implement and evaluate architectures for supporting this process using services and aspects.

2.1 Interaction Specifications with Services

Analyzing the requirements leads to a number of use cases and roles. The roles relevant for our example are *AwareClient* (weather-aware clients), *Manager* (drives the update process), *Broadcaster* (broadcasts messages to a group of clients) and *Arbiter* (collects responses from clients).

We specify the services of the CTAS weather update system using a notation based on Message Sequence Charts (MSC) [8, 13, 31]. An MSC defines the relevant sequences of *messages* (represented by labeled arrows) among the interacting *roles*. Roles are represented as vertical axes in our MSC notation. Figures 3(a) to 3(d) show the specification of several services as interaction patterns and Fig. 3(e) shows the roles and their connections. The MSC syntax we use should be fairly self-explanatory, especially to readers familiar with UML2 [31]. In particular, we support labeled boxes in our MSCs indicating alternatives and conditional repetitions (bounded and unbounded loops). Labeled boxes *on* an axis indicate actions, such as local computations. High-level MSCs (HMSCs) indicate sequences of, alternatives between and repetitions of services in two-dimensional graphs – the nodes of the graph are references to MSCs, to be substituted by their respective interaction specifications. HMSCs can be translated into basic MSCs without loss of information [13].

A number of extensions to the standard MSCs warrant explanation [15, 17]. First, we take each axis to represent a *role* rather than a class, object, or component. The mapping from roles to components is a design step in our approach and will be described in detail in Sect. 2.2. Second, axes labeled with an asterisk refer to *all* entities "playing" this particular role at runtime – this allows us to model broadcasting succinctly with MSCs; the MSC in Fig. 3(d) shows an example: the *Broadcaster* role sends the *Msg* message to *all* entities that at runtime play the *AwareClient* role. Third, we use an operator called *join* [13, 15], which we use extensively to compose *overlapping* service specifications. We call two services *overlapping* if their interaction scenarios share at least two roles and at least one message between shared roles. The join operator will *synchronize* the services on their shared messages, and otherwise result in an arbitrary



Arch	CTASMgr	CTASBroadcast	CTASArbiter	CTASClient
1	Manager, Arbiter, Broadcaster			AwareClient
2	Manager, Broadcaster		Arbiter	AwareClient
3	Manager, Arbiter	Broadcaster		AwareClient
4	Manager	Broadcaster	Arbiter	AwareClient
5	*Manager, Forwarder*	*Broadcaster*	*Arbiter*	*AwareClient*

Figure 3: CTAS Services, Roles, Role Domain Model, and Architecture 2

interleaving of the non-shared messages of its operands. Join is a powerful operator for separating an overall service into interacting sub-services. The availability of such an operator also distinguishes our approach from many others in literature.

For reasons of brevity, we omit the specification and description of the full set of services for the CTAS weather update cycle in this paper. For the full set, see [21].

2.2 Architecture Definition

The next step after eliciting the services is to define a suitable component architecture that can provide these services. We must make sure that the architecture observes the dependencies of the roles and further constraints given by the requirements. Our goal is to explore multiple alternatives of such architectures for their adequacy in supporting the elicited services. Interesting questions to ask in evaluating and deciding on a target architecture include (among many others) (a) to what degree are the components of the architecture coupled, and (b) what are the implications of tight and loose coupling on latencies for calls between components? For this paper, we will focus on these questions for the architecture exploration we conduct; however, the method we propose supports exploration of a wide variety of architectural properties.

For our example, we initially define four architectures that differ in the component configurations and the roles played by the components; see Fig. 3(f). A fifth architecture, shown in row 5 of this table, will be discussed in Sect. 4.

A component can play multiple roles. Intuitively, "playing a role" in an architecture means implementing all interactions in which this role partakes. The more roles it plays, the more functionality it implements. Interactions between different components usually mean expensive distributed communications, while interactions between roles within one component can be implemented very efficiently as subroutine or method calls. This decision, as well as the number of actual component instances deployed, makes the difference when comparing architecture alternatives.

The table in Fig. 3(f) shows the roles played by the components in the various architectures. A blank cell means that the corresponding component does not exist for that architecture. CTASClient plays the role of an AwareClient in all architectures. CTASMgr also exists in all architectures, but the roles it plays differ. The components CTASArbiter and CTASBroadcaster exist depending on whether CTASMgr will play the role of Arbiter or Broadcaster, respectively. In Architecture 2 of Fig. 3(g), for instance, CTASMgr plays two roles: Manager and Broadcaster.

We have introduced a dedicated Architecture Definition Language – the Service-ADL [17, 20] – that captures services, roles and components for the precise specification of component architectures. We make use of Service-ADL specifications in subsequent steps.

3. TRANSLATING ARCHITECTURES TO ASPECTS

We show now how to map the previously identified services (i.e. their interaction patterns) to different deployment architectures. We define how to translate a service-based system specification into aspect-oriented programs [11]. The basic idea is to translate the interaction patterns defining the services into aspects so that they can be woven into any given component configuration using AspectJ's weaving capability. We thus use AspectJ to separate structure from behavior and later combine both using the AspectJ compiler (weaver).

3.1 Aspect-Oriented Programming: AspectJ

AspectJ [10, 33] is a general-purpose aspect-oriented extension to the Java programming language; its language constructs facilitate clean modularization of separate concerns. AspectJ provides a compiler that weaves aspect code into Java classes.

We translate our service model into aspects using AspectJ's *join points*, *pointcuts*, *advice*, *aspects*, and *intertype declarations* [33]. Examples of join points are method calls, method executions, object instantiations, constructor executions, field references and handler executions. Pointcuts are used for selecting these join points; an example of a pointcut is "all invocations of method xyz". Advice defines code that executes before, after or around a pointcut. An aspect

can be the combination of a pointcut and the corresponding advice. In other words, using pointcuts, an aspect can specify at what points in the execution – or under what circumstances – a particular piece of code, represented as an advice, should be called. An intertype declaration, can be used to specify a set of members (attributes, methods) that should be present in multiple classes. We use pointcuts and advice to translate patterns of interactions defining a service as an aspect; we use aspects describing intertype declarations to implement associations between roles and components.

3.2 Translation Process and Artifacts

The translation process has two phases (cf. Fig. 4, within the dashed boundaries): (1) implementing a common service repository based on a set of identified roles, and (2) implementing multiple architectures for the common service repository. In (1) we use a build file to weave together the following artifacts: classes for roles, aspects implementing the associations in the role domain model, aspects introducing the methods and local operations that each role needs to support, and aspects that implement the interaction pattern of each service. In (2) we weave together the output of (1) with classes for the components and aspects implementing the associations of the component configuration. These steps are explained, in detail, below.

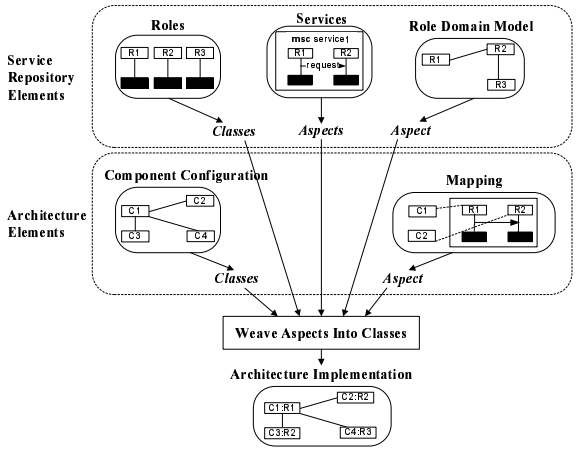


Figure 4: Implementation Process Artifacts

3.3 Translating Roles

Define classes for roles: For each role appearing in a service definition we create a class. All role classes are derived from a base "role" class, which has attributes representing the role's state and parent component name. The parent component will capture the name of the component for which the role instance is playing the role.

Define aspects for the role domain model: We capture the associations between roles as specified in the role domain model in form of attributes of the created role classes. Fig. 3(e), for instance, indicates that the Manager role needs to have a reference to a Broadcaster role. Thus, a new attribute of type Broadcaster and corresponding accessor methods are introduced to the Manager role class with the help of an intertype declaration. All these relationships are captured in one aspect representing the configuration for the roles, cf. Fig. 5.

```
public aspect CTASRoleConfiguration {

        private Broadcaster Manager.broadcaster;

        public Broadcaster Manager.getBroadcaster() {
                return broadcaster;
        }

        public void Manager.setBroadcaster(Broadcaster b) {
                broadcaster = b;
        }
        ...
}
```

Figure 5: Role Domain Model Aspect

3.4 Service Repository

For each service to be supported by the architecture, we follow the steps described below.

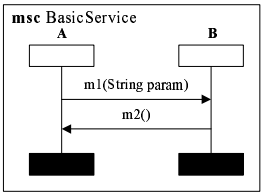
Define aspects for role interactions and local operations: For each possible received message of a role we introduce a method into the role class using an intertype declaration. For the service shown in Fig. 3(c), we introduce the method CTAS_GET_NEW_WTHR for the role AwareClient. We do this for all messages of the service and also for all local operations of a role.

Define aspects for services: So far we have defined classes for all roles and connected them as specified in the role domain model. We have provided methods within each role class for all possible incoming messages and local actions. We now define each service as an aspect; we use pointcuts and advice to connect messages and local actions according to the service specification within the MSC. Service specifications can make use of the full expressive power we provide in our extended MSC dialect. This includes sequential and parallel composition, alternatives, loops and join composition, as explained in [15, 13]. In the following, we explain the translation of each of these operators into aspects.

We define the service in terms of an aspect with the help of pointcuts and advice. The *next* operation or message sent within an interaction pattern is implemented as an advice for the pointcut defined for the *current* interaction. A series of these definitions enables us to coordinate the interactions in the implementation. In essence, this corresponds to capturing a global state machine for each service in terms of an aspect definition – this is the basis for automating the translation from MSCs to aspects as we will discuss in more detail in Sect. 3.7.

Sequential Composition: In Fig. 6, we show how an MSC with simple sequential ordering of messages is translated to an aspect. We identify pointcuts for the occurrences of the method (received message) and define advice for this pointcut. Here, we define a pointcut called *Interaction_m1(B b)* which captures the method call *m1(String)* for the targets of type role B. We define an *after* advice for this pointcut which executes after the method call. The advice defines the call of the method *m2()* of role A. Thus, the coordination of the interactions for this MSC is achieved with the help of the aspects defined using pointcuts and advice. If we need to change the order of the interactions, all we do is update this one aspect without modifying any other piece of code.

Consider, for instance, the service shown in Fig. 3(c). We define a pointcut for the CTAS_GET_NEW_WTHR message receipt by an AwareClient role. The next step of the interaction, executing the local operation of Get New Weather,




```
public aspect BasicService {

 pointcut Interaction_m1(B b):
  target(b) && (call( void m1(String) ));

 after (B b) : Interaction_m1(b) {
        A a = getARole();
        a.m2();
  }
}
```

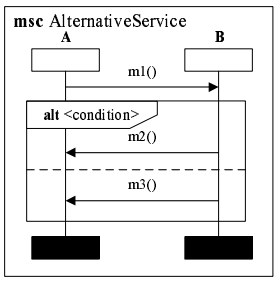
Figure 6: Translation of Basic MSC

is implemented as an advice for the pointcut just defined. The aspect defined for this service is shown in Fig. 7.

```
public aspect ServiceClientGetNewWeather {

 pointcut Interaction1(AwareClient ac):
  target(ac) && (call(void CTAS_GET_NEW_WTHR()));

 after (AwareClient ac) : Interaction1(ac) {
    ac.GetNewWeather();
 }

 pointcut LocalOperation1(AwareClient ac):
  target(ac) && (call(boolean GetNewWeather()));

 after (AwareClient ac) returning (boolean flag) : LocalOperation1(ac) {
  ac.getManager().CTAS_WTHR_RECEIVED(ac,flag);
 }
}
```

Figure 7: Service represented as an Aspect

Alt: The *alt* operator within an MSC specifies alternative interactions as given by the two operands. The first alternative is chosen if the specified condition yields true; otherwise the second alternative is chosen. The translation for an MSC that uses the *alt* operator to define a service is shown in Fig. 8. Note that for an automated translation, the condition must be specified in a format that can be translated to a Java boolean expression.



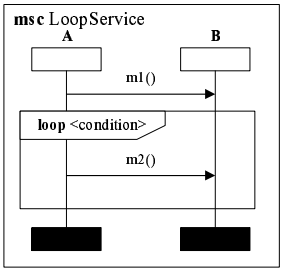

```
public aspect AlternativeService{

 pointcut Interaction_m1(B b):
  target(b) && (call( void m1() ));

 after (B b) : Interaction_m1(b) {
        A a = getARole();
        if (cond)
                a.m2();
        else
                a.m3();
  }
}
```

Figure 8: Translation of MSC with alt

Loop: The *loop* operator within an MSC requires the occurrence of the given interactions as long as the specified condition yields true. For an MSC with the *loop* operator, the advice for the pointcut implements the loop. The MSC and its translation is shown in Fig. 9.

A variant of the *loop* operator exists for loops with a defined number of repetitions. In this case, the number of repetitions is indicated by an integer value instead of a condition. Translation into an advice is straightforward. Instead of a while-loop, a for-loop with the specified repetition counter value is used.

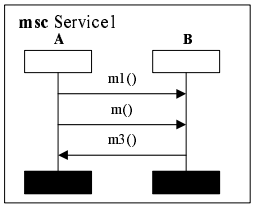
Join: The *join* operator is an extremely powerful means to combine and synchronize overlapping services – it is central in our approach. Recall that we call services overlapping if they share at least two roles and at least one message between shared roles. *join* synchronizes its operands on shared messages, while imposing no ordering on all others; in other words, a join is the parallel composition of its operands, with the restriction that the operands synchronize on shared messages (cf. middle part of Fig. 11). Interactions that are shared in both services will occur only once in the resulting service. This means, that all interactions causally before a shared interaction within *both* services must have occurred before the shared interaction can itself happen. Note, that the *join* operator does not change the order of interactions in any of the operands. It only restricts the occurrence of shared messages. For a formal definition of the join semantics, see [13, 15].



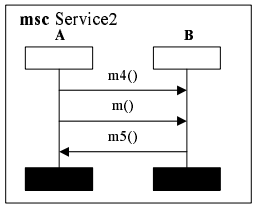

```
public aspect LoopService{

 pointcut Interaction_m1(B b):
  target(b) && (call( void m1() ));

 after (B b) : Interaction_m1(b) {
        A a = getARole();
        while(condition)
                a.m2();
  }
}
```

Figure 9: Translation of MSC with loop

The translation of services joined using the *join* operator requires a high degree of coordination. For the join, we need to synchronize the interactions around common messages. We also have to ensure that the overlapping interactions only happen once. To achieve this, we define conditional variables that help us track the interactions.




```
public aspect Service1{
 pointcut Interaction_m1(B b):
  target(b) && (call( void m1() ));
 after (B b) : Interaction_m1(b) {
        m();
 }
 pointcut Interaction_m(B b):
  target(b) && (call( void m() ));
 after (B b) : Interaction_m(b) {
        A a = getARole();
        a.m3();
 } }
```




```
public aspect Service2{
 pointcut Interaction_m4(B b):
  target(b) && (call( void m4() ));
 after (B b) : Interaction_m4(b) {
        m();
 }
 pointcut Interaction_m(B b):
  target(b) && (call( void m() ));
 after (B b) : Interaction_m(b) {
        A a = getARole();
        a.m5();
 } }
```

Figure 10: Translation of Operand MSCs for the join Operator in Fig. 11

In Fig. 10, we define two services Service1 and Service2 where the message *m()* is common to both of them. We have to ensure that when these two services are joined as specified in Fig. 11 there is only one occurrence of this message. We also have to ensure that both messages *m1()* in Service1 and *m4()* in Service2 occur before *m()* can occur. The actual series of interactions that happen when the two services are joined is shown in the center of Fig. 11. The two con-

ditional variables *m1* and *m4* in the translation keep track of the occurrence of the respective messages. The purpose of using join is to provide the developer with the means to independently define two services and subsequently be able to compose them as shown in Fig. 11. This decouples the service definitions and enables us to powerfully compose complex system behavior by joining separate, easy to understand services.

Thus, the join of the two services holds back the message *m()* until both messages *m1()* and *m4()* have been received. This is done by specifying the advice "around" a pointcut. The "around" advice is executed instead of the pointcut, and the pointcut is executed with the help of the *proceed* call only if the join conditions are satisfied. The aspect for ServiceJoin updates the flags when *m1()* and *m4()* are received, and triggers *m()* when both flags are set, see right part of Fig. 11.

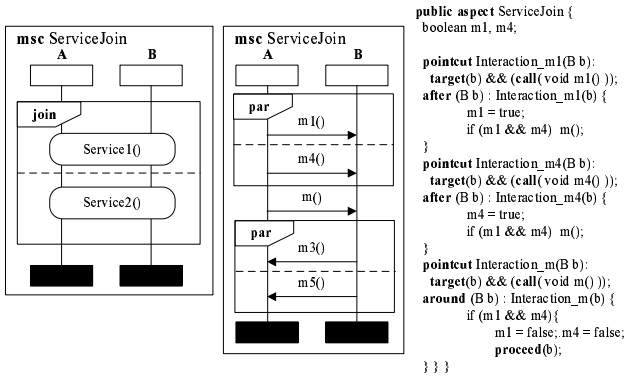


Figure 11: Translation of MSC with join

The shared messages of both services need to be identified by the aspect implementer. As we will show in Sect. 3.7, however, this step can be automated within a tool. Note also that the join operator permits more than two operands. In this case, the translation procedure as given above generalizes straightforwardly.

MSC references and HMSCs: MSC references represent "calls" to services from within an MSC specification. Fig. 11 shows an example. The join operator contains references to the two joined services in the operands. MSC references allow us to avoid redundancy and to provide a simple structuring mechanism for MSCs. Because the referenced MSCs can be fully expanded into the referencing MSC, the translation to aspects is covered by the explanations above. High Level MSCs (HMSCs) (cf. Fig. 3(a)) simplify graphical specification of sequencing, alternatives and loops. In [13], we have given a transformation algorithm for HMCSs into basic MSCs. This transformation is straightforward albeit tedious for more complex HMSCs. The translation of the resulting basic MSCs again follows the procedure as defined above.

Define build file for service repository: Based on the roles and services we created, we now define a build file that lists all classes and aspects in order to provide a complete role implementation.

3.5 Translating Components

We can now define multiple component configurations for the system in development. We create different component types according to the roles they play in specific architectures.

Define classes for component types: We define one class for each component type in a specific configuration. These classes are the same and can be reused for other configurations that make use of the same component types.

Define aspects for each component configuration: We establish a specific mapping of roles to a component type by introducing attributes into the component type classes. We do this again with the help of intertype declarations. We define one aspect for each component configuration to reflect the roles the components play in that configuration. After the weaving process the component classes will contain fields with references to the roles they play – together with accessor methods – and thus provide access to the roles' implementation.

3.6 Defining the Architecture

To finally establish a specific architecture, we create a build file that selects the build file for the service repository and the classes and aspects for a specific component configuration. As a consequence, we can create multiple configurations by defining multiple architecture build files which differ *only* in the classes and aspects selected for a component configuration. The code for the services remains unchanged.

3.7 Towards an Automation of the Translation Procedure

In the previous sections we have defined a precise and unambiguous procedure to translate a service model into executable prototypes. While we have geared the presentation to supporting a manual translation, the procedure is suitable to be automated by a tool – provided, of course, that the service model is present in a machine-readable format. Such a tool removes most of the manual effort from the implementers and enables rapid, cost-effective generation of architecture exploration prototypes. Multiple architecture alternatives can be evaluated with very little effort within a short time. In the following we will describe a tool we are developing that fulfills these requirements and allows us to generate customized AspectJ implementations of service specifications.

The aspect generator makes use of parts of the tool chain described in [16]. The starting point for this chain is a modeling tool for interaction specifications, called *M2Code*. M2Code uses Microsoft Visio as graphical front-end for editing MSCs and HMSCs to model service specifications. Behind the scenes, M2Code captures the modeled MSCs from Visio in form of an integrated interaction-based data model. M2Code has algorithms for automatically translating MSCs into state machines [14] – these state machines are saved by M2Code together with the rest of the interaction model into an XML file; this XML file is the input for the M2Aspects generator for AspectJ code. M2Aspects is the component implementing the procedure as described above. In addition to the service specifications in form of MSCs, it also requires the specification of target architecture configurations in form of Service-ADL files. Currently, the user has to manually provide these files. We are working on creating a user interface for role to component mappings and several alternative architecture configurations. Once invoked, M2Aspects creates all Java classes, aspects and AspectJ build files required

for a compilable and runnable prototype. Fig. 12 shows the aspect generation tool and an overview over necessary input and created output artifacts.

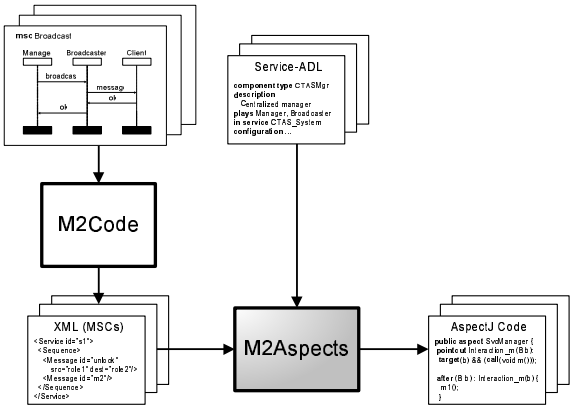


Figure 12: Aspect Generation Tool Chain

The AspectJ prototype code generator can be invoked several times for several different architecture configurations without having to change the service repository. This supports a comparative exploration of alternative component architectures that provide the same set of services to the environment. We will show in Sect. 4 how the prototypes can be further enhanced with performance-measuring code for evaluation – AspectJ is used here as well.

4. EXPERIENCES AND EVALUATION

We have applied our approach extensively by evaluating and optimizing different architectures for the CTAS case study, as documented in [20]. Our architecture exploration included several steps. First, we have evaluated the first four architecture alternatives shown in Fig. 3(f) by applying the above described translation procedure to create executable prototypes out of service models. We have defined new services to perform performance measurements to evaluate the prototypes. Our approach made it very easy to weave these services as aspects into the existing source code without changing any of the existing services.

The above exploration lead to the assumption that a different role set, incorporating a Forwarder role mediating between Manager and Broadcaster, is more effective in reducing the execution time required to perform a weather update cycle – a critical part of the system. We followed our process of Fig. 2 and iteratively evolved the architecture, role and service specifications. This was very simple and fast because we could apply the changes to our strictly decoupled service model and the similarly separated AspectJ implementation artifacts. The new architecture variant turned out to be most efficient. We finally explored systematically multiple instantiations of this architecture with different numbers of Forwarder components. By measuring the performance of different prototypes we could optimize the component configuration to find the "sweet spot".

For performance evaluation, we measured absolute elapsed time as well as logical communication latency using the notion of logical clocks [18]. Evaluating absolute times as well as relative latency values helped us to abstract from the communication infrastructures used. We could select architecture configurations that are optimized in terms of communications overhead and that perform similarly well in concrete deployments on specific messaging infrastructures. Details, statistics and performance charts are documented in [20].

Overall, the translation procedure resulted in the following numbers of artifacts: we created six Java classes for the roles (including the base class), one aspect defining the role configuration, eight aspects defining the services of the CTAS weather update system, four classes containing implementations of the local actions, 16 classes representing the different messages between the roles and one service repository build file. All these files were created once and remained unchanged in the subsequent steps. For different architecture variants to compare, we created four classes defining the component types, one aspect defining the component configuration (mapping of roles to components) and one build file listing the full set of files to be woven together to create the executable prototype. Component configuration and build files were different for each architecture variant; the component type classes could be reused. The performance evaluation code required the introduction of three new role classes and four new service aspects into the service repository.

The overall code size was 115 KB of Java and AspectJ code. We organized the different types of source code files in an intuitive directory and package hierarchy. Creating and exploring different architecture variants was very easy and cost effective. An analysis shows that the five different architectures could be generated with significantly less effort as compared to writing each prototype from scratch. The ease of "weaving-in" analysis code allowed us to perform measurements on the architectures that would have been difficult and cumbersome to achieve for five separately coded prototypes. Additionally, the created source code remained manageable when service repository changes were necessary. Modifications could be kept local and contained which reduced the potential of unintended side-effects or errors. Overall, this together with the improved understandability contributed significantly to an efficient prototyping and evaluation process.

5. RELATED WORK

Our approach is related to the Model-Driven Architecture (MDA) [22] and architecture-centric software development (ACD) [31]; similar to MDA and ACD we also separate the software architecture into abstract and concrete models. In contrast to MDA and ACD, however, we consider services and their defining interaction patterns as first-class modeling elements of *both* the abstract and the concrete models. Furthermore, we do not apply a transformation from abstract to concrete model. Our work is related to the work of Batory et al [28]; we also identify collaborations as important elements of system design and reuse. Our approach in particular makes use of MSCs as notation and is independent from any programming language constructs.

We consider services as aspects in the sense of AOP [11] at the modeling level. In Aspect-Oriented Modeling [5], cross-cutting concerns are captured as *design* aspects. In we model interaction patterns that span multiple components, the services, as aspects. In the tradition of [32] the role concept is also adopted in [5] to define aspects abstractly. [12] discusses role models and their properties in object-orientation. The work in [7] makes use of roles to capture design patterns using aspects. An early approach to map

collaboration-based designs with roles to AOP can be found in [24]. All of these approaches, however, lack the "join operator" that we use to compose *overlapping* services based on shared messages.

Often, the notion of service-oriented architectures is identified with technical infrastructures for *implementing* services, including the popular web-services infrastructure [29]. Our work, in contrast, supports *finding* the services that can later be exposed either as web-services, or implemented as "internal" services of the system under consideration. Because our entire approach is interaction-based it is perfectly general with respect to the types of architectures we can model – in this paper we have modeled, for instance, a distributed version of the Observer pattern [6], other architectural styles such as pipes and filter and layered architectures [27] can be captured in the same way by extracting their characteristic interaction patterns and use cases.

Our work is further related to the early performance testing approach of [3], which makes use of the early availability of off-the-shelf components, and tests performance relevant interactions of critical use cases. Similar to our approach, [3] concentrates on interactions. Our approach differs in that it is fully model-based and allows for automatic generation of testing prototypes. The PPCB framework and modeling approach [19] aims at performance predication with the help of benchmarking key architectural elements. The so gained results improve the performance prediction model for the distributed application. The authors apply statistical methods to analyze the prediction results and check accuracy.

An alternative approach for architecture exploration to the one presented is to generate different prototypes directly from the model using a code generator. In [16], we presented a tool chain that allows us to generate executable service-based systems from service models. While this approach provides excellent possibilities for conformance testing and immediate deployment on different target environments, we found it less effective for comparative architecture exploration. The flexibility of creating source code that separates the different system services from the possible target architecture configurations and from performance measurement code provides the developer with a powerful toolset for efficient architecture exploration. Furthermore, the full flexibility that source code provides remains.

6. DISCUSSION

Our translation from partial interaction specifications in form of MSCs into AspectJ code can be viewed as "aspects on steroids" in the sense that it translates each service into an AspectJ aspect. At first sight this indicates a major limitation of this approach: the final code emerges only from the weaving of all classes containing structural information and all aspects capturing role configurations, interactions, and local actions; the complete picture of the behaviors of each individual component is only contained in the resulting Java bytecode. Clearly, this prevents easy refinement on the *component* level. Recall, however, that we set out to develop our approach for efficient and effective *architecture* exploration, where a lot depends on the cross-cutting interaction rather than on the detailed, fine-grained behavior of individual components, which can be developed after settling on one component configuration. We see our approach most effective for exploring different choices on the architectural level; the services as captured for the architecture exploration can, of course, still inform the implementation of the final deployment architecture.

In general, there may be interactions between different aspects; in other words the advice provided by multiple aspects might affect the same pointcut. This might lead to unexpected states or results, for instance when two services are joined in our implementation. However, such unwanted interactions can already be detected on the level of interaction patterns in our service model and are thus best handled on this level, not on the level of aspects. The tool chain we have referred to in Sect. 3.7 has elements for formal verification of service interactions that resolve this problem [16].

The approach we present here improves upon our earlier work in [17], by using pointcuts and advice to represent interaction patterns. This provides better decoupling between the roles and the interaction patterns they participate in as compared to the class-based approaches in [23] and [9].

7. CONCLUSIONS AND OUTLOOK

Thorough exploration of architectural alternatives is particularly important for complex distributed and reactive systems. However, tight coupling between the domain logic and the implementation infrastructure, as well as prohibitive costs for building prototypes needed to evaluate *multiple* architectures often are stumbling blocks for architecture exploration.

In this paper we have shown how to define software architectures, describe measurements and explore architecture alternatives using the notion of services and their embodiment as aspects in AspectJ. Services are partial interaction specifications of systems. We have decoupled the services provided by the system from the many target architectures that can implement the same set of services. We have introduced a translation process turning service-oriented architecture specifications into AspectJ aspects. This process exploits AspectJ's weaving capability to map service specifications to target architectures. We have shown how to automate this procedure and described our tool chain to generate AspectJ prototypes from service models specified using our Service-ADL.

The same translation process can also be used for services describing measurements used in the evaluation of an architecture. AspectJ's weaving joins the aspects needed for the evaluation with those for the core functionality. Consequently, the evaluation services are decoupled from the rest of the architecture; the architecture does not need to change for the purposes of the evaluation. The examples we presented also demonstrate how easy it is to iteratively change a given architecture – only a subset of the service repository needed to be modified to fundamentally change the broadcasting architecture for the CTAS case study we used as a running example. Performing these changes and the subsequent exploration was a matter of minutes in the given system. The example we used is a representative for many similar systems from database systems to business information systems using web services as communication mechanism between distributed components.

Future work will include enhancing the automated translation from MSCs to aspects, and investigation of the relationship between our technique for architecture exploration with runtime verification techniques, such as [26]. Another avenue for investigation is how the generated aspect code can be consolidated into a fewer number of files once a com-

ponent architecture has been chosen. This will allow us to use the generated AspectJ code even better as an evolutionary prototype instead of a valuable, cheap but mostly explorative prototype.

8. ACKNOWLEDGMENTS

Our work was partially supported by the UC Discovery Grant and the Industry-University Cooperative Research Program, as well as by funds from the California Institute for Telecommunications and Information Technology (Calit2). Further funds were provided by the Deutsche Forschungsgemeinschaft (DFG) within the project *InServe*. We are grateful to the anonymous reviewers for insightful comments.

9. REFERENCES

[1] M. Broy and I. H. Krüger, editors. *Pre-Proceedings of the Automotive Software Workshop San Diego 2004*. UCSD, 2004. `http://aswsd.ucsd.edu/2004`.

[2] P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures – Methods and Case Studies*. Addison-Wesley, 2002.

[3] G. Denaro, A. Polini, and W. Emmerich. Early performance testing of distributed software applications. In *WOSP '04: Proceedings of the 4th international workshop on Software and performance*, pages 94–103. ACM Press, 2004.

[4] E. Evans. *Domain Driven Design*. Addison-Wesley, 2003.

[5] R. France, G. Georg, and I. Ray. Supporting Multi-Dimensional Separation of Design Concerns. In *The 3rd AOSD Modeling With UML Workshop*, 2003.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[7] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of OOPSLA'02*, pages 161–173. ACM Press, 2002.

[8] ITU-TS. Recommendation Z.120 : Message Sequence Chart (MSC). Geneva, 1996.

[9] E. Kendall. Aspect Oriented Programming for Role Models. In *International Workshop on Aspect Oriented Programming at ECOOP*, volume 1743 of *LNCS*, pages 294–295. Springer Verlag, 1999.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, volume 2072 of *LNCS*, pages 327–353. Springer Verlag, 2001.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect Oriented Programming. Technical report, Xerox Corporation, 1997.

[12] B. Kristensen. Object-Oriented Modeling with Roles. In *Proceedings of the 1st Conference on Object Information Systems*, 1996.

[13] I. Krüger. *Distributed System Design with Message Sequence Charts*. PhD thesis, Technische Universität München, 2000.

[14] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to Statecharts. In F. J. Rammig, editor, *Distributed and Parallel Embedded Systems*, pages 61–71. Kluwer Academic Publishers, 1999.

[15] I. H. Krüger. Capturing Overlapping, Triggered, and Preemptive Collaborations Using MSCs. In M. Pezzè, editor, *FASE 2003*, volume 2621 of *LNCS*, pages 387–402. Springer Verlag, 2003.

[16] I. H. Krüger, J. Ahluwalia, D. Gupta, R. Mathew, P. Moorthy, W. Phillips, and S. Rittmann. Towards a Process and Tool-Chain for Service-Oriented Automotive Software Engineering. In *Proceedings of the ICSE 2004 Workshop on Software Engineering for Automotive Systems (SEAS)*, 2004.

[17] I. H. Krüger and R. Mathew. Systematic Development and Exploration of Service-Oriented Software Architectures. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 177–187. IEEE, 2004.

[18] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 27(7):558–565, July 1978.

[19] Y. Liu and I. Gorton. Accuracy of Performance Prediction for EJB Applications: A Statistical Analysis. In *Proceedings of the 4th Int. Workshop on Software Engineering and Middleware (SEM'2004)*, volume 3437 of *LNCS*, pages 185–198, 2005.

[20] R. Mathew. Systematic Definition, Implementation and Evaluation of Service-Oriented Software. Master's thesis, University of California, San Diego, 2004.

[21] R. Mathew and I. H. Krüger. Full Service Specification for CTAS System, 2006. `http://sosa.ucsd.edu/publications/icse2006/CTASServiceSpecification.pdf`.

[22] OMG Model Driven Architecture. `http://www.omg.org/mda`.

[23] B. Paech. A Framework for Interaction Description with Roles. Technical Report TUM-I9731, Technische Universität München, 1997.

[24] E. Pulvermüller, A. Speck, and A. Rashid. Implementing Collaboration-Based Designs Using Aspect-Oriented Programming. In *TOOLS '00: Proc. of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00)*, pages 95–104. IEEE Computer Society, 2000.

[25] SCSEM 2003 Case Study. 2nd Int. Workshop on Scenarios and State Machines: Models, Algorithms, and Tools. CTAS Case study Overview, Requirements, 2002. `http://www.doc.ic.ac.uk/~su2/SCESM/CS/requirements.pdf`.

[26] K. Sen, G. Rosu, and G. Agha. Runtime Safety Analysis of Multithreaded Programs. *Proceedings of the 10th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 337–346, 2003.

[27] M. Shaw and D. Garlan. *Software Architecture, Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[28] Y. Smaragdakis and D. Batory. Implementing Layered Designs with Mixin Layers. In *Proceedings of ECOOP 1998*, volume 1445 of *LNCS*, pages 550–570. Springer Verlag, 1998.

[29] J. Snell, D. Tidwell, and P. Kulchenko. *Programming Web Services with SOAP*. O'Reilly, 2002.

[30] D. Trowbridge, U. Roxburgh, G. Hohpe, D. Manolescu, and E. Nadhan. *Integration Patterns. Patterns & Practices*. Microsoft Press, 2004.

[31] UML 2.0. `http://www.omg.org/uml`.

[32] M. VanHilst and D. Notkin. Using Role Components to Implement Collaboration-Based Designs. In *Proceedings of OOPSLA'96*, pages 359–369. ACM Press, 1996.

[33] Xerox Corp., Palo Alto Research Center Inc. The AspectJ Programming Guide. `http://www.eclipse.org/aspectj/doc/released/progguide`, 2004.

[34] P. Zave. Feature-Oriented Description, Formal Methods, and DFC. In *Proceedings of the FIREworks Workshop on Language Constructs for Describing Features*, pages 11–26. Springer-Verlag, 2001.