Systematic Development and Exploration of Service-Oriented Software Architectures

Ingolf H. Krüger and Reena Mathew
Computer Science and Engineering Department
University of California, San Diego
9500 Gilman Drive, La Jolla, CA 92093-0114, USA
{ikrueger,rmathew}@ucsd.edu

Abstract

The notion of service is becoming increasingly popular as a means for implementing large-scale distributed, reactive systems. Systematic development approaches and modeling notations for services are still largely missing from the literature. We introduce an Architecture Definition Language for service-oriented software architectures. It provides modeling elements for interaction patterns defining services, as well as for mapping sets of services to target component configurations. We also present a comprehensive software development process that considers services as first class modeling elements. By decoupling the modeling of services from their implementation on target component configurations this process enables exploration of multiple architectures implementing the same set of services. We substantiate our view of services as cross-cutting architectural aspects by providing a mapping from services to aspects in AspectJ. We illustrate applicability of our approach by modeling service-oriented architectures for portions of the Center TRACON Automation System as a running example.

1. Introduction

Complex, distributed and reactive systems are increasingly composed of a multitude of services; these services emerge from the interplay of several components collaborating to complete a desired task. Current software development approaches and modeling notations focus mainly on individual components instead of on the interaction patterns defining services as cross-cutting system properties.

We present an approach to designing software architectures starting from service specifications. To that end, we introduce an interaction-based service notion together with a definition language for service-oriented software architectures. Exploiting the cross-cutting nature of services we also show how aspect-oriented programming languages, such as AspectJ [13], can be used to explore different deployment architectures for a given set of services. This is an important asset for validating architectural sustainability.

1.1 The Notion of Service

Services play an increasingly important role as a modeling and implementation concept across application domains. The telecommunications domain has long embraced the notions of features and services as the main constituents of its products. Web services have emerged in the business domain as an attempt at simplifying distribution, publishing, discovery, addressing, and accessing software functions across the Internet. Even in the automotive domain [25], representative for the area of large-volume embedded systems with high dependability requirements, the notion of service is catching on as a means for addressing the ever increasing distribution and ensuing complexity of automotive software.

Despite the importance and prevalence of services, however, no systematic, methodological approach to service-oriented software development exists to date. In addition, none of the prominent modeling notations currently available addresses the service concept as a first-class modeling entity. Informal definitions for the term service abound (cf., for instance, [27,32,31,28,11]); these definitions, however, typically capture only syntactic lists of operations upon which a client can call instead of the interaction patterns behind them. This is inadequate as a starting point for systematic service-oriented development.

In our view, a service is defined by the interaction among the entities involved in establishing the service.
Defining services by the interactions that ensue from invoking them goes well beyond the predominantly syntactic service notions cited above: it provides a handle at meaningful concepts for service composition, refinement, and validation, and introduces them as first-class modeling elements as opposed to being merely first-class implementation elements. In [17] we have shown how our service notion can be readily formalized using the mathematical model of streams; here, however, we focus on methodological aspects of service modeling and implementation.

1.2 An Architecture Definition Language for Services

The definition of an adequate software architecture is one of the decisive steps in the development of complex, distributed, and reactive systems. Traditionally, the components of the system under consideration and their relationships (also sometimes called interfaces) are deemed important ingredients for the definition of software architectures [2]. As the “glue” between components, connectors are introduced [1], encapsulating the protocol required for a particular architectural style [30]. An architectural configuration is a connected graph of components and connectors that describes the composition of these elements into a system.

In this paper we argue that the interaction patterns defining services delivered by a software system are cross-cutting aspects of the corresponding architecture, and should be represented explicitly on the level of architectural definition languages (ADLs [24]). A central theme of our approach is to define services as building blocks for the architecture. In particular, we are interested in exploring multiple architectural configurations that implement a given set of services.

To that end, we introduce an ADL that defines services as interaction patterns among a set of roles. Each role is a placeholder for a concrete component in an interaction pattern. Components typically participate in multiple services. For instance, a particular component may act as a server in one service, while it acts as a client in another. Roles make explicit the different configurations in which a component can participate in service executions. At any point in time during execution of the system under consideration a component can play certain roles, such as the client or server role in a client/server interaction. A component can play more than one role, and can change the roles it plays over time. Our ADL provides means for associating concrete components with the roles they play to yield an architectural configuration.

To capture the interaction patterns among the roles of the system we use an extended version of Message Sequence Charts (MSC) [10,16]. MSCs have proven useful for defining interaction protocols graphically. They form the basis for interaction models in the most recent rendition of the UML [34]. We introduce two extensions to the MSC notation we need for service specifications: (1) each axis represents a role instead of a concrete component, (2) we treat each syntactic element in an MSC (role names and messages) as parameters that can be substituted to form instances of interaction patterns [19]. We will discuss these extensions in more detail in the context of concrete examples, below.

1.3 Service-Oriented Development

For service-oriented system development we suggest to follow a systematic development process as outlined in Figure 1.

Figure 1. Service-Oriented Software Architecture Development

This iterative process mainly involves two phases: (1) Define the set of services of interest — we call this set the service repository; (2) Map the services to component configurations to define deployments of the architecture. Phase (1) starts by identifying the relevant use cases and their relationships in the form of a use case graph. From these use cases the roles and their interactions are derived as defining elements of services. This gives rise to a domain model for the roles involved. 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 again may give rise to a fresh look at the use cases.

1.4 Contributions and Outline
In the remainder of this text we explain the various elements of our ADL for service-oriented software development together with the development process outlined above in more detail. To illustrate concepts and notations we introduce a realistic running example in Section 2. In Section 3 we describe the notational elements of our ADL for defining roles, services, and components. In Section 4 we apply the mentioned development process to the running example, and show how to yield multiple different architectural configurations for the same set of services. In Section 5 we discuss the exploration of the resulting architectures by introducing a mapping to AspectJ; this mapping exploits the weaving operator to map roles to architectural configurations. Section 6 contains a discussion of related work. Our conclusions and outlook appear in Section 7.

2. Example – CTAS case study

To illustrate the central challenges in specifying services, as well as the applicability of the service concept introduced above, we model part of the Center TRACON Automation System (CTAS) using an extended service notation. This case study has been studied extensively recently in the context of scenario modeling and state-machine synthesis [29,6]. CTAS is a set of tools and processes for air traffic management at large airports. According to [29] a key element of CTAS is the continual distribution of accurate weather information to client processes. This information is received by a central communications manager process, and then forwarded to all client processes that have successfully registered with the communications manager. Examples for clients are processes for airplane route analysis, aircraft and weather panels, as well as graphical user interfaces for simulation purposes.

We will use a simplified version of the CTAS system for illustrating the usage of our ADL. Specifically, we will consider the process of updating all the clients interested in receiving weather forecast updates. The main components of the system are the Communications Manager (CM) and the various types of clients that are notified of the latest weather information. The central property of the weather update logic is that all clients depending on weather data (weather-aware clients) should have (and use) a consistent view on the weather information.

The centralized CM is responsible for updating the clients. Based on the requirements document [29], the CM has to broadcast messages to all the weather-aware clients for a weather update. The clients respond with a yes or no message (indicating that their attempt at receiving the weather update has succeeded or failed, respectively); CM has to decide how to proceed with the update phase depending on whether all the clients respond yes or at least one responds with no. If all clients respond yes then the CM notifies all of them to use the new weather information sent to them. If one of the clients responds no, then all clients are instructed to continue to use the previously distributed weather information. This ensures that all clients update the weather data simultaneously and consistently. In the following sections we will identify the key architectural entities and their interactions to deliver the weather update service for CTAS.

3. Roles, Services, and Components as Architectural Elements

In this section we introduce the conceptual and syntactic elements of our Architecture Definition Language (ADL) for defining service-oriented software architectures; in particular, we work with roles, services (defined as interaction patterns of roles), and components implementing services. In Section 4 we will see how these elements combine for defining the architecture of CTAS. However, for illustration purposes we will refer to elements of the case study already in this section.

3.1 Roles

Components typically participate in multiple services. For instance, a particular component may act as a server in one service, while it acts as a client in another. To make the different configurations in which a component can participate in service executions explicit, we introduce the concept of a role. At any point in time during execution of the system under consideration a component can play certain roles, such as the client or server role in a client/server interaction.

Within CTAS we can, for instance, identify roles for the clients of the central communications manager: they can be aware or unaware of weather updates. Clients that are aware have successfully registered with the communications manager, and participate successfully in each update cycle; clients that have registered unsuccessfully, or that fail during the update cycle are (or become!) unaware. As illustrated in this example, components can change roles over time; this distinguishes roles clearly, for instance, from the class concept in object-orientation. Semantically, roles map to predicates over the state space of the system under consideration.
Roles are defined in terms of their interactions and the various states they can be in along the way. As mentioned above, roles, their interaction patterns and corresponding states are determined from the use case graph as part of the requirements capture process. Often, this process allows us to immediately determine state names to be associated with roles even before interaction patterns are defined.

Syntactically, we introduce roles (more precisely: role names) using the following template:

```
role <name>
description <description>
states ( <statename> <description> )*  
```

Each role has a unique name. The description section can be used to give a short explanation of the role’s purpose. The states section lists states the role can be in while executing the services of the system.

The roles defined using this template will be mapped to components of a concrete architectural configuration. For CTAS the roles we can immediately identify are the Manager, Broadcaster, Arbiter and two types of clients: AwareClient and UnawareClient. These entities interact with each other to execute the service of updating all clients with the latest weather information.

### 3.2 Services

As mentioned above, we define services as the interaction patterns required to establish a specific task. This identifies services as partial behaviors of the system under consideration; components, in contrast, are total behaviors. To describe the interaction patterns defining services we use an extended version of Message Sequence Charts (MSC). MSCs have emerged in the context of SDL [8] as a means for specifying communication protocols in telecommunication systems. They have also found their way into the new UML 2.0 standard [34], which significantly improves the standing of interaction models within the UML. In the notational variant we consider here, each MSC consists of a set of axes, each labeled with the name of a role. An axis represents a certain segment of the behavior displayed by its corresponding role. Arrows in MSCs denote communication. An arrow starts at the axis of the sender; the axis at which the head of the arrow ends designates the recipient. Intuitively, the order in which the arrows occur (from top to bottom) within an MSC defines possible sequences of interactions among the depicted roles.

Services associate roles with interaction patterns. The template for defining a service is given below:

```
service <name>
description <description>
interaction <Message Sequence Chart>
```

An MSC shows how the roles interact to establish the service under consideration; it can also contain information about the states the roles are in during the course of the interaction. To express that a message is sent to all components playing the same role at a given point in time, we adorn the outlined rectangle at the top of the role axis with a “*” and leave it empty otherwise; in Figure 2 we use the label “< |*>” to denote the choice between the empty string and “*”.

![Figure 2. MSC Template](image)

If there is a transition from one role to another during the course of an interaction pattern we indicate this by a filled rectangle followed by the name of the new role on the corresponding axis.

![Figure 3. MSC with Role Transition](image)

Figure 3 shows an example: after receiving the first message, the right hand side roles changes to another one.

Analyzing the CTAS system, we can see that there are interactions between the entities for broadcasting messages and collecting responses from the clients. The weather update process makes use of these interactions to update all the clients. These interactions will be defined using the service notion in Section 4.1.

### 3.3 Components

Components are the architectural entities implementing roles. Each component may play multiple roles. The mapping from roles to components determines the services a component participates in and,
thus, also the component’s interaction behavior. Because there may be multiple components implementing the same set of roles in a concrete architecture, we work with the notion of a component type. The template for defining a component type is as follows:

```
component type <name>
description <description>
plays (<rolename>)*
in service (<servicename>)*
```

Instances of the type play the roles listed in the `plays` section. The `in service` section defines the services in which these roles are played. Actual component instances are defined to be of a particular component type. Syntactically, this is shown as follows in an architecture definition (see Section 4.2):

```
<component instance name> : <component type>
```

Clearly, there are multiple ways for mapping a set of services to components. Each such mapping defines an architectural configuration.

4. Service-Oriented Development

In the following paragraphs, we elaborate on the steps required to define an architecture for CTAS, and illustrate the ADL templates used to define the elements of the architecture.

4.1 Service Elicitation

In the first step, we determine the relevant use cases of the system under consideration. These use cases can be related to each other by virtue of inclusion and extends relationships [34]; we can express these relationships in terms of a use case graph – this graph will also help determine the relationships between services elicited from the use cases. For simplicity we consider the following two use cases for CTAS: Broadcast Message and Update Clients. The relationship between these use cases is shown using the graph in Figure 4.

```
Figure 4. CTAS Use Case Graph
```

By analyzing the use cases, for which a detailed description appears in [29], we are able to identify the entities that interact with each other to establish the use cases. This determines the roles of the system under consideration. For CTAS we can identify the following roles: Manager (the central communications manager), AwareClient (for clients participating successfully in the weather update process), UnawareClient (clients not participating successfully), Broadcaster (for distributing messages to all aware clients), and Arbiter (for collecting the response from all aware clients after sending out new weather information). Figure 5 shows the specification for these roles.

```
role Manager
description Manages the initialization and updates of all clients
states done, pre-initializing, initializing, post-initializing,
pre-updating, updating, post-updating, post-reverting

role AwareClient
description Weather Aware Clients of the CTAS System
states pre-initializing, initializing, post-initializing,
pre-updating, updating, post-updating, post-reverting

role UnawareClient
description Weather Unaware Clients of the CTAS System
states

role Broadcaster
description Broadcasts messages to a set of components
states

role Arbiter
description Accumulates client responses to determine an outcome
states all_yes, some_no
```

```
Figure 5. Roles for CTAS system
```

We have also listed state names that can be readily determined by reading the requirements document (cf. [29]). Both the Manager and AwareClients go through a cycle of pre-initializing, initializing and post-initializing (to register clients), and pre-updating, updating, and post-updating (to handle weather information transmission). The post-reverting state is reached after recovering from an unsuccessful weather update. The Manager also has a “done” state. The Arbiter’s states “all_yes” and “some_no” capture whether all clients have responded with “yes” or at least one has responded “no”, respectively.

The transitions between these states are captured as part of the interaction patterns elicited next to define services. In our example, there are messages being broadcast to a set of clients – this defines the “broadcast” service. There are interactions required for collecting all the responses from the clients and evaluating those to yield a single response – this defines the “arbiter_collect” service. These interactions are integrated to yield the “update” service. These services and their defining interactions are shown in Figures 6, 7 and 8. Recall that in our extended MSC notation we use role axes labeled with a “*” to refer to all compo-
nents playing that role. Consequently, in an implementation of the service, the message “receiveBroadcast” is sent to all components playing the AwareClient role (in arbitrary order).

**service** broadcast
**description**
Broadcasts messages to a set of clients
**roles**
Manager, Broadcaster, AwareClient
**interaction**

![Figure 6. Broadcast service](image)

**service** arbiter_collect
**description**
Determine outcome based on client responses
**roles**
Arbiter, AwareClient, UnawareClient
**interaction**

![Figure 7. Arbiter collect service](image)

Similarly, in the “arbiter_collect” service, all AwareClients each send any one of the following two messages back to the Arbiter role: receiveClientResponse (YES) or receiveClientResponse (NO). In the latter case, AwareClients become UnawareClients. In Figure 8 we also use MSC references (labeled, rounded rectangles) to call upon other services; these references are resolved similar to a “macro expansion”. If m and n are message labels, then we write “P(m/n)” to indicate substitution of m for n in the target MSC P; this replaces all occurrences of n in P by m. Similarly, if s and t are roles, then “P(s/t)” indicates substitution of s for t in P. For example, the “broadcast” service could have been called in the “update” service using broadcast(AwareClient/Receiver). Labeled hexagons denote states of roles in the course of an interaction. If no explicit label is given, the state of the corresponding role is unspecified and unconstrained.

**service** update
**description**
Updates all the clients with the latest weather updates
**roles**
Manager, Broadcaster, Arbiter, AwareClient
**interaction**

![Figure 8. Update service](image)

The collection of all services elicited in this way can be considered as a service repository for the system. Once we have defined the service repository, the relationships between the different roles of the system are clear. We express the structural part of these relationships in the form of a domain model. This domain model allows us to identify the dependencies between the different roles of the system and is used later on to identify and define the component configurations for the architecture. The domain model for the CTAS roles is given in Figure 9.

![Figure 9. Model for CTAS Roles](image)

### 4.2 Architecture Definition

Following service elicitation, we can define the architecture for the system under consideration. There can be many possible component configurations for a system that implements the elicited services. Therefore, we need to first define the component configuration for which we need to define architectures. Such configurations can be based on various architectural styles [30] such as pipes and filters, layered architecture, or client-
Component configurations can be defined based on the architectural style the system is intended to adhere to.

**architecture CTAS1WeatherUpdateArchitecture**

**description**
Updates weather clients with latest weather information

**component types**
- **component type CTASMgr**
  **description**
  Centralized manager for the CTAS system
  **plays** Broadcaster, Manager, Arbiter
  **in service** update

- **component type CTASClient**
  **description**
  A type of weather client
  **plays** AwareClient
  **in service** update

**components**
CM:CTASMgr, PGUIClient:CTASClient
RAPClient:CTASClient

**configuration**

We represent component configurations using a deployment domain model. The components defined in this domain model receive their behavior definitions by mapping the services onto the chosen component configuration. In Figures 10 and 11 we show two component configurations based on the client-server architectural style.

Once a component configuration has been defined, the architecture can be determined by selecting the services from the service repository that need to be supported by the architecture and mapping each service to the component configuration and defining the component types. This happens according to the following two steps: (1) Select a service to be supported by the system, (2) Map the roles of the service to the component types supporting that service. These steps need to be repeated for all services to be supported by the system. The mapping step can be performed manually, or – because we are working with MSCs – (semi-) automatically, using algorithms for state machine synthesis, such as [20].

**architecture CTAS2WeatherUpdateArchitecture**

**description**
Updates weather clients with latest weather information

**component types**
- **component type CTASMgr**
  **description**
  Centralized manager for the CTAS system
  **plays** Manager, Arbiter
  **in service** update

- **component type CTASBroadcaster**
  **description**
  Broadcasts messages to a set of components
  **plays** Broadcaster
  **in service** update

**components**
CM:CTASMgr, CMBroadcaster:CTASBroadcaster
PGUIClient:CTASClient, RAPClient:CTASClient

**configuration**

We now define two architectures for CTAS which differ in their component configuration. Both architectures will support the functionality of updating all the weather clients with the latest weather information. In other words, both support the service “update” from the service repository. The notation (service-name)* in the implements section of the architecture indicates that the service can be executed multiple times in sequence.

**Architecture 1**: For the first architecture, the component configuration CTAS1 in Figure 10 is assumed. When we select the service “update” and do the mapping we obtain the architecture shown in Figure 10 by mapping the Manager, Broadcaster and Arbiter roles to the component type CTASMgr and the AwareClient role to the component type CTASClient.

**Architecture 2**: Figure 11 shows a second component configuration and corresponding architecture; here, the
definition for CM has changed as the new component CMBroadcaster takes on the role of the Broadcaster. The definitions for the clients remain unchanged.

### 4.3 Refinement And Refactoring

In practice, the initial architecture determined using any process, including the one outlined above, will have to be refined as more information about the services and target component configurations is revealed. This is supported by the iterative nature of our process, allowing the architect to refine the architecture by collecting increasingly detailed information about the use cases, roles, and services of the system under consideration. Traditionally, refactoring [9] is a form of code modification, used to improve the software structure in support of subsequent change and long-term maintenance. Our ADL and supporting process support refinement and refactoring on an architectural level by allowing the architect to modify the component configuration to yield new architectures: the service elicitation step is decoupled from the architecture mapping. Consequently, both the service repository and the target architecture can be modified independently of one another before the next architectural mapping step is performed.

### 5. Architecture Exploration in AspectJ

One of the central goals in introducing our ADL for service-oriented systems is to provide a platform for exploring various alternatives for implementing a set of services on different target architectures. In this section we show how this can be accomplished by identifying services and the interactions they are based on as cross-cutting aspects of an implementation. Based on this intuition we exploit the “weaving” capability of AspectJ [14] to join the abstract role descriptions contained in the ADL with implementations of the deployment domain models to yield concrete implementation architectures. Aspects [14] can be considered as system properties that cross-cut components. Aspect oriented programming (AOP) is a programming technique that makes it possible to clearly express aspects including appropriate isolation, composition and reuse of aspect code. AOP cleanly separates components and aspects from each other by providing mechanisms that make it possible to abstract and compose them to produce the overall system. With the help of AOP, we can implement the system by defining services as aspects.

We realize this idea by associating with each role of the architecture a separate Java class. Similarly, we introduce a Java class for each component type of the architecture. Services become “aspects” in that they determine the “glue” between components and the roles they implement.

![Figure 12. Implementation Process](image)

Because the role specifications are independent of any concrete target architecture this enables fast exploration of architectural configurations. In fact, the implementation of the architecture can be automatically generated once the specifications for the roles, services and components are defined. The process of implementing an architecture based on the idea outlined above is shown in Figure 12.

Technically, the service aspect for a component introduces a new attribute pointing to an instance of the role the component has to play for this service. This mapping of roles is based on a slightly modified approach explained in [12]. In this case we are defining an implementation in Java with the help of aspects using AspectJ [13]. For reasons of brevity, we do not explain the entire implementation; instead, we sketch the framework for the solution.

### 5.1 Components and Roles

Component types are mapped to concrete classes that will be used for implementing the system. The definition of a component type and its attributes will be based on the deployment domain model defined for a specific architectural configuration. The entities in this deployment domain model can be defined as standard Java classes.

Roles can be defined as classes in Java as well. The methods of the class corresponding to a role will be based on the interactions defined in the services this role is a part of. In fact, because we have based our definition of services and roles on interaction patterns and MSC specifications we can use automaton synthesis algorithms (such as [20]) to automatically derive the behavior of role classes from service specifications. The resulting roles can be associated with multiple components. The roles and components defined for our CTAS example are shown in Figure 13.
5.2 Aspects

As defined above, component classes alone do not have any of the properties of the roles. We introduce aspects that glue the components with the roles they support. Each aspect introduces the roles as attributes of the components that play the role. In our running example, we introduce a new attribute of type “Broadcaster” called “broadcaster” for the class CTASMgr. Analogously we introduce an “awareClient” attribute for the CTASClient component type. The definition of the corresponding aspect is shown in Figure 14.

The next step is to ensure that the components are equipped with the methods required for supporting the relevant services. In this example we consider only one service, namely broadcast, in Architecture CTAS1 (cf. Figure 11). This involves introducing a new method for the respective component types depending on the services supported. The method is, in fact, a wrapper method to the actual method call to the corresponding method in the role class. For instance, a new method called broadcast is introduced in the CTASMgr component class. This method is implemented by delegating the call to the role that supports this method; in this case, this is the broadcaster role.

Once the aspects are defined as outlined in the preceding paragraph, they can be woven into the component and role classes to generate an implementation for the architecture. Thus, multiple implementations can be generated by enabling or disabling the aspects to be woven into the component and role classes.

6. Related Work

Rapide [22], C2 [33], Darwin [23] and Wright [1], are among the more prominent ADLs. These ADLs differ in how they define components, connectors and configurations. For example, Rapide models connectors implicitly, whereas C2 and Wright have explicit connectors [24]. In comparison to these ADLs, we concentrate on defining services as first-class, cross-cutting modeling entities independent of actual components/connector configurations. In fact, each connector and component configuration of [33,1] can serve as the basis for a deployment domain model onto which we can map the elicited services. Our approach can be viewed as a methodological front-end to implementation-oriented notations for service specification [4,35,21].

Stepwise refinement [7] of software architectures is advocated in [3], where features are composed in a functional, algebraic style. In contrast, our services cross-cut the components of a deployment domain model, and thus defy the concept of pure function composition. Instead, we can use (semi-)automated procedures for generating state machines from role specifications [20], and automatically obtain multiple
mappings of these to different component configurations.

In our ADL we explicitly address important elements of service specification, including the possibility of components changing their roles over time, and communication from and to all components playing a certain role at a given moment. Modeling notations such as the UML, even in its most recent version [34], show much potential for improvement in this regard.

Our approach to service modeling is related to the role concept introduced in [26] and the activities of [15]. While our service concept is based on interaction patterns, stressing the cross-cutting nature of services, the roles of [26] describe projections of such patterns onto individual components; to yield the overall picture the latter have to be recomposed into a global interaction specification. Activities of [15] capture global interaction properties as we do in our service definition; in contrast to our approach, however, [15] views activities as classes and roles as extensions to these classes. We have adopted the more general and abstract viewpoint of defining services as interaction patterns that can be mapped to any target software architecture.

7. Conclusion and Outlook

Across application domains, the notion of service has emerged as a means for structuring complex distributed and reactive systems. Systematic development processes and modeling notations for service-oriented software and systems engineering are, however, still largely missing from the literature, let alone industrial practice.

In this text, we have introduced an Architecture Definition Language (ADL) together with a systematic development process for service-oriented software architectures. The ADL captures services as interaction patterns among a set of roles. Each role models the behavior a concrete component must display to participate in the implementation of a service. Our use of roles in the definition of the service notion decouples services from target component configurations and architectures. In fact, we can define mappings from a set of services onto a given target architecture by appropriately associating roles with components implementing their interaction behaviors. This allows exploration of multiple “candidate” architectures implementing the same set of services. We have shown how this form of architectural exploration can be substantiated using the aspect-oriented programming paradigm of AspectJ; we have mapped roles and components to Java classes, whereas services bind these two together in the form of corresponding aspects.

The systematic service-oriented development process we have introduced proceeds in two phases. First, the set of relevant services is elicited, starting from a set of use cases for the system under consideration; together with this set of services a role domain model is built, indicating the structural relationships between the roles participating in the execution of a service. Second, the role domain model is refined into a component configuration, and the elicited services are mapped onto this component configuration. Our process is iterative in nature, supporting refinement and refactoring of the services and domain models.

We have applied our process and notation to the Center TACON Automation System; we have shown portions of it as a running example in this text. Our observation is that our approach scales well even for large numbers of services; both role and component domain models can be used to define hierarchical architectures.

Future work will include providing largely automated tool support for refinement and refactoring of services and component configurations, as well as for architecture mappings. Comparing our approach to component-oriented ones using metrics [5] adapted to the notion of service should provide further insight into the utility of services as an element of both modeling and implementation. Also, the relationship between services and aspects is an interesting area for further exploration.

Acknowledgements

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. We are grateful to Manfred Broy and the anonymous reviewers for insightful comments.

References


