Dealing with Active and Stateful Services in the Service-Oriented Architecture*

Haldor Samset and Rolv Bræk

Department of Telematics, NTNU,
N-7491 Trondheim, Norway
{haldors,rolv}@item.ntnu.no
http://www.item.ntnu.no

Abstract. Services in SOA are typically considered to be of passive nature, providing functionality that solely execute upon invocation. Additionally, stateless services are commonly advocated as a modeling principle of todays SOA style.This paper argues that services could be of an active nature, and that services often involve sessions with stateful behavior. We suggest an approach for modeling active and stateful services , using UML 2 Collaborations and state machines. This forms a behavioral contract, and separates the modeling of service logic from the service implementation, allowing for validating the asserted service behavior using a model checker.

Keywords: Service-oriented architecture, service modeling, collaboration-based, stateless, stateful, behavioral contract.

1 Introduction

With its raising popularity, the Service-oriented Architecture (SOA) approach has started to find its way into to the telecom domain. Traditionally, software engineers within the telecom domain have applied the abstraction of communicating state machines as the basis for modeling of communication systems and services. This has helped considerably to manage their inherent complexities. In the information systems domain, where focus has been more on computation and retrieval of data, procedure calls have formed the main behavioral abstraction. Now that the two domains are merging one may ask if these differences are fundamental or accidental? We believe there is a fundamental difference between passive services and active services, i.e. services with active behavior that takes initiative towards the service user. Such services are typical for the telecom domain. This is due to the fact that they involve several active users that may take initiatives towards each other, and also that behavior is time dependent. Notification and push services are also active. The most challenging situation occurs when the service and the user may take mutual initiatives.

* This paper is partly supported by the SPICE project, IST Contract No. 027617.

E. Di Nitto and M. Ripeanu (Eds.): ICSOC 2007 Workshops, LNCS 4907, pp. 268–281, 2009.

2 Service-Oriented Architecture

The service-oriented architecture (SOA) is essentially an architectural style where program functionality is logically organized into services that are universally accessible through well-defined interfaces, supported by mechanisms to publish and discover the available functionality and the means of communicating with the service. Ideally, a service should provide a rather self-sufficient piece of functionality applicable within a particular context. The organization of functionality into services is meant to give rise to a larger degree of functionality reuse. This is achieved by keeping service interfaces independent from functionality implementations, and by enabling universal access and keeping them independent from other service interfaces.

Contemporary[1] SOAs have primarily been client-server oriented, with clients making synchronous calls to operations provided by a server. Although there is some movement towards using asynchronous events with the emergence of "advanced SOA"[2] [13], symmetrical asynchronous communication between peers seems to be the exception rather than the rule.

2.1 Principles in the Service-Oriented Architecture

An architectural style include some principles or rules that guide and constrain the construction of systems according to the architecture. For SOA, these principles are related to the concept of service-orientation. While there is no common agreement on exactly what these principles are, the "four tenets of service-orientation" by Don Box of Microsoft [1] are commonly cited in the SOA community. The four tenets are stated below in bold, followed by a short explanation.

Boundaries are explicit. Services are geographically separated, developed by different organizations, run in a variety of execution environments etc. All these distributions bring some aspect of boundaries, and the potential cost of crossing these should be explicit. The notion of boundaries also applies to the service modeling itself, where the size and complexity of abstractions shared between services should be kept to a minimum.

Services are autonomous, in the sense that they are independent, both as deployable units and as self-governing entities. There is no central controlling entity in a service-oriented system, rather services are responsible for their own behavior.

Services share schema and contract, not class. Unlike the combination of structure and behavior in the class construct of object-orientation, service interaction is based on the concepts of schema and contract. The contract describes the structure and ordering constraints of the messages belonging to the service, and the schema describes the conceptual data manipulated.

[1] As Erl[4], we use *contemporary SOA* to refer to the typical WSDL- and Web Services-based service-oriented architectures usual in the industry today.

[2] Or "SOA 2.0" which seems to be a popular, but somewhat misleading term as there is no version numbers or standardization as such of SOA.

Service compatibility is based upon policy. According to [1], structural and semantic compatibility is often confused in object-oriented design. Structural compatibility in service-orientation is based on schema and contract, while semantic compatibility is based on policies, stating the conditions and assertions for the desired service behavior. Service Level Agreements (SLAs) would typically be part of such policies.

3 Passive and Active Services

Telecom architectures are characterized by peer-to-peer services with multi-way initiatives[6], as opposed to the client-server, one-way initiatives of contemporary SOA services. Our notion of active services is an attempt to broaden the SOA service concept to incorporate these characteristics.

3.1 Passive Services

We consider services in contemporary SOAs to be passive in the sense that the functionality they provide is considered to be executed solely upon invocation. In this context, the term *service request* is typically applied to describe the invocation of an operation available in a declared interface of a service. The service concept in SOA has grown out of the Web Service technology, and is tightly tied to the concept of interface as declared by WSDL [12], which is restricted to describing the static invocation details of interfaces. In this sense, the client-server paradigm of HTTP pervades the service model of SOA.

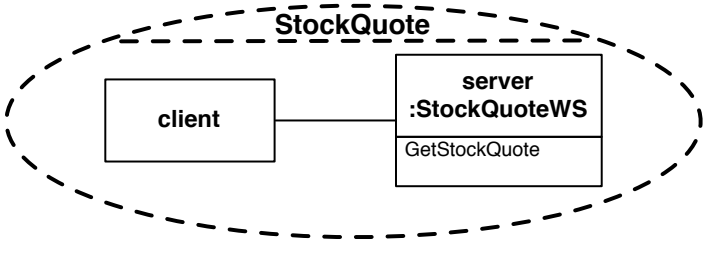


Fig. 1. StockQuote Web Service collaboration

If we view a typical HTTP Web Service in the light of UML 2 Collaborations [15,23], we see that such a service is the collaboration between a client and a server. Only the server role is described in any detail (by being typed by an interface), while the client role remains anonymous. Figure 1 shows the classic StockQuote Web Service depicted as a collaboration, with the server role typed by the StockQuoteWS interface. The only active part in the collaboration is the client role, invoking the provided operations on the server. Interactions are thereby limited to initiatives in one direction only.

3.2 Active Services

A rich class of services are of an active nature, in the sense that their functionality is not only invoked by an client, but is active behavior that may take initiative towards the client, seemingly on their own. The classic telephone service is a good case in point as a user may both initiate and receive calls. Likewise, the numerous scenarios describing advanced context-aware services performing proactive behavior exemplify active service functionality that is not inherently passive. In general, services involving several active participants (humans or devices) that may take independent initiatives to interact are of an active nature. Active objects with concurrent behavior do not communicate well by synchronous invocation, since this involves transfer of control, and blocking behavior[6]. Some sort of asynchronous communication mechanisms such as signal sending or messaging is most convenient for active service interactions.

The participants of active services cannot well be classified as "clients" or "servers", and thus they do not fit naturally into the client-server paradigm of contemporary SOA. It is possible to design workarounds for this mismatch, but they are artificial and invariably add complexity for the service design.

4 Services and State

In the SOA literature one frequently encounters the statement that *services should be stateless*. Erl, for instance, lists this among his service-orientation principles in [4]. This statement is routinely justified by referring to the stateless nature of the HTTP protocol, demonstrating the common assumption of SOA as a direct derivative of the Web Service model. A service could be viewed as completely stateless if all operations declared by that service could be invoked at any time, providing the same result regardless of any invocation order. But more often than not, an order in which the operations should be invoked exists - like when parameters of an operation need values obtained by an earlier invocation of another operation. Hence, a service is usually not entirely stateless.

However, states are mostly relevant for individual service sessions. In many SOA applications, each interface handles many concurrent sessions. This reflects itself in the typical conversation-identification parameters of contemporary SOA services, revealing the occurrence of a session beneath the surface. Although each session may be state dependent, the interface definitions may appear to be stateless, while in reality they just hide the sessions and their states. For instance, in many cases a client has to authenticate and authorize itself against a provided service, and a session identification has to be carried in all subsequent service requests. If the client fails to initiate a new request within a certain timeframe, the session will be invalidated and further requests rejected until new authentication and authorization is obtained. Since it is the stateful session behavior that really matters for clients, it should be represented clearly to handle the complexities in a controlled way. We therefore argue for enabling explicit modeling of individual service sessions and their behavior. If this happens to be

stateless, so much the better. If it is stateful, it should be made explicit as this is essential to validate interfaces and manage service composition.

People tend to worry about solutions being stateful, and their main concerns are usually related to performance. The rather simplistic view that stateless solutions scale and stateful does not, leads many to believe that the notion of state should be avoided. This is not true, there are no performance problems associated with states, as it is demonstrated by existing high performance communication software. There is no problem today to automatically generate such software for state machine models [3].

The question is not whether to treat states, but how and where. In contemporary SOAs state is not treated explicitly, but rather hidden in the implementation details of services and their consumers. We contend that one is better off dealing with stateful behavior as explicitly as possible. This can be achieved by modeling the stateful behavior separately from the implementation, and defining a precise mapping from the model to the implementation.

5 Modeling Active Services in SOA

So how do we model our active services while still aligning ourselves with the service-oriented architecture style? A central point in SOA is the notion that services are requested and provided. The terms *requestor* and *provider* are often applied when talking about the entities that respectively request and provide services [4,5]. (OASIS, on the other hand, uses *service consumer* and *service provider* in its reference model [14]). These terms are usually assumed to relate to the client and server roles in the client-server communication paradigm.

In contrast to the common view in contemporary SOA, we focus on the collaborative nature of services. Even in the simple case where the functionality of a service is a mere computation or information lookup, both the component requesting the computation and the component performing the computation are involved in the execution of the service in our view. We regard a service as *an identified (partial) functionality, provided by a system, component, or facility, to serve a purpose for its environment* [2]. Hence, we maintain the idea that services are something that are provided, while still allowing for modeling services as a compromise of several collaborating entities. Consequently, we focus on describing the actual collaboration of these entities, in other words the interactions between the collaborating entities. We believe that this is where the main complexity of the service design task resides.

We choose to keep the terms *requestor* and *provider* as the identified roles in a service in the context of SOA, but we dispose of the imposed client-server relationship. Accordingly, we view a service in SOA as the collaboration between two entities that are equal peers when it comes to the ability to take initiative (a *composite* service may involve more than two entities, but this is outside the scope of this paper). The roles requestor and provider describe the characteristics that the entities must possess to take part in this collaboration. With the client-server relationship exchanged for a peer-to-peer paradigm, it may seem

that the names requestor and provider have no significance. However, the role names do imply a difference. The requestor role describes the entity that takes the very first initiative of the service, in other words the entity that is in need of a certain functionality. The provider role correspondingly represents the behavior of the entity providing the requested functionality. Provider role implementations would most likely be the targets of lookup and discovery, like the provided services in contemporary SOA.

5.1 Modeling with UML 2 Collaborations

The new Collaboration concept introduced with UML 2.0[15] provides us with a suiting modeling element for our service notion. By being both a structural and behavioral classifier, it fits well for modeling the collaboration between a requestor role and a provider role. The signals a role should be able to receive, as well as which signals it should be allowed to send, are of interest when describing the service behavior. A CollaborationRole in UML 2 Collaborations can be typed with an interface, so we type roles with interfaces describing the possible signal receptions in order to constrain the reception. The constraint on possible signal sending from a role is given by the possible receptions defined by the other role in the collaboration. But this is only the static communication details, we also want to describe the exact interaction protocol. For this we choose to model the *visible* behavior of each role using state machines. The state machine model is a UML 2 state machine limited to simple triggers on events of type SendSignalEvent or ReceiveSignalEvent. A role state machine then describes the different possibilities of behavior allowed by a role in different states - it can either receive or send signals.

We have defined a UML profile for modeling active SOA services with UML Collaborations. Figure 2 shows the stereotypes which is the core of this profile.

The BasicService stereotype is a Collaboration with two collaboration roles, named requestor and provider. We decided against naming it ActiceService, since the stereotype is more general and could be used to model both active and passive services. In the latter case, all initiatives would taken by the requestor role, but this would not demand a separate stereotype. The name BasicService also indicates that we foresee a stereotype for composite services at a later stage, with more than two roles involved. A BasicService has two ServiceRoleStateMachines, describing the (visible) behavior of the requestor and provider roles.

We impose the following restrictions on the stereotypes in Figure 2:

1. A transition in a ServiceRoleStateMachine must be a ServiceRoleTransition,
2. The trigger of a ServiceRoleTransition is limited to events of type SendSignalEvent or ReceiveSignalEvent. This restricts the transitions to model only the sending or reception of a Signal. Timeouts are to be notified using signals representing the timeout events.
3. A ServiceRoleTransition can not specify any constraints; there may be no guards, preconditions and postconditions.

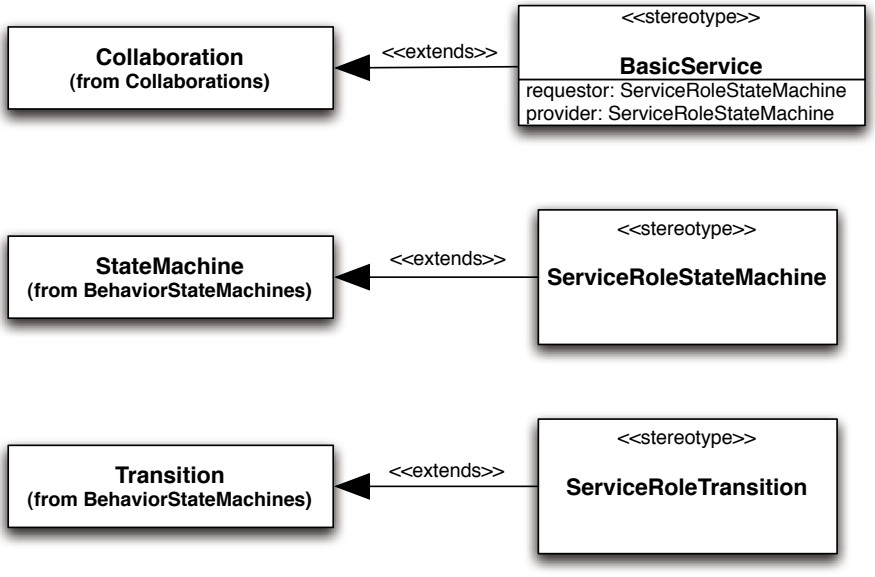


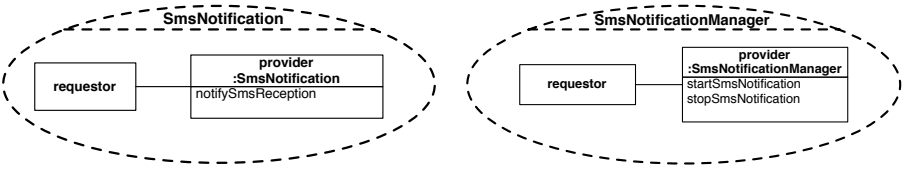
Fig. 2. Profile

4. The only PseduoStateKind allowed is *initial*, and there must be exactly one.
5. A ServiceRoleStateMachine must have exactly one region with one initial state.

5.2 Example

Let us consider an example inspired by the Short Messaging Service (SMS) functionality exposed by the Parlay X specification [7]. Even though not primarily designed for SOA, this standard is intended to make telecom functionality like the SMS and call control available for 3rd party applications produced by non-telecom developers. It does so by specifying a collection of Web Services that is typical in a contemporary SOA.

One of the Web Services described by the Parlay X specification is the SmsNotification service. Its purpose is to provide other components with the ability to receive incoming SMS messages from the network. A short number (or a keyword combination for a short number) is reserved and registered with the operator prior to using the service. A component that wants to utilize this functionality must implement the Web Service specified by the SmsNotification interface. In addition it has to provide the URL of the Web Service, together with the criteria for notification to the Parlay X server, in advance. Related to this service, the SmsNotificationManager Web Service allows components that receive SMS messages via SmsNotification to start and stop the delivery of messages. This



(a) SmsNotification service (b) SmsNotificationManager service

Fig. 3. Parlay X SMS Notification Web Services

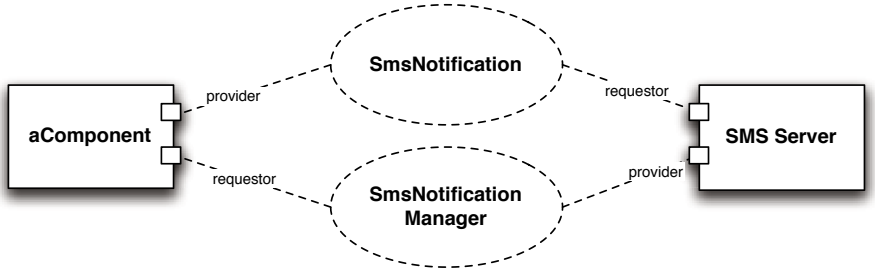


Fig. 4. Parlay X SMS Notification Web Services applied to components

service is to be implemented by the network component managing the SMS resources. Figure 3 shows these services as collaborations, while Figure 4 shows, with collaboration uses, how they are applied in a component system.

SmsNotification and SmsNotificationManager obviously contain related functionality, and they are certainly not independent of each other. One can argue that they are so strongly related that the functionality of these services should preferably be gathered in a single service, but due to the passive nature of Web Services this is not possible. If we presently dispose of the Web Service paradigm at our abstract modeling layer, and adapt the idea that services indeed can be active, these services can easily be combined into one single service, as shown in Figure 5.

We have developed an Eclipse plugin for modeling active services as UML 2 Collaborations in our integrated service engineering tool suite Ramses [10]. An initial attempt to model the role behaviors of the active SmsNotification service would typically result in the state machines shown in figure 7 (where ! denotes a SendSignalEvent and ? a ReceiveSignalEvent). The requestor role requests to be notified of incoming SMS messages, and receives notifications until it informs the provider that it wants to stop receiving.

Modeling the interactions with state machines allow us to validate the service behavior. We have developed another Eclipse plugin that generates a Promela model, based on the two role state machines contained in the active service collaboration. This enables us to analyze the behavior using the Spin [8] model checker, which quickly reveals that NotifySmsReception signals can be lost. As the

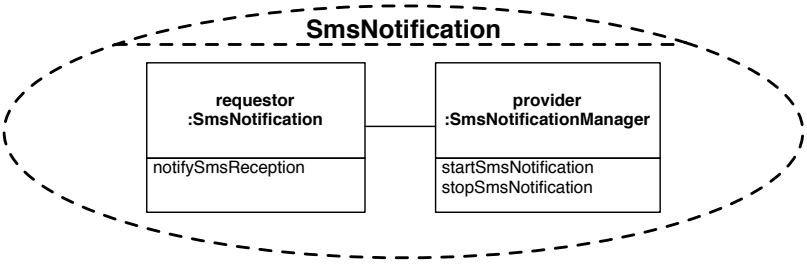


Fig. 5. New active SmsNotification service

Fig. 6. Active SmsNotification service applied to components

requestor is allowed to end its behavior immediately after sending the StopNotification signal, the provider could manage to send one or more NotifySmsReception signals before the StopNotification signal is received. This is the problem of *mixed initiatives*, a common situation that occurs when a role is in a state where it can both receive or send a signal, and is easily fixed once recognized. Figure 8 shows the corrected state machines, where the provider now must acknowledge that it will stop the notification by sending an AckStopNotification signal. Correspondingly, the requestor has to receive any incoming NotifySmsReception signals until it receives the acknowledgment.

5.3 Model and Implementation

We have separated the service modeling from implementation by applying the UML 2 Collaborations for modeling, deliberately removing ourselves from implementation technology details like WSDL. In order to provide implementations of the services, the model must be precisely mapped to an execution environment and code must be generated or written accordingly. There is not room for demonstrating such a mapping here, but the active service collaboration concept conforms to our system engineering approach based on a meta-model for design and execution of services. Central in this model is precise behavioral descriptions of components as communicating extended finite state machines in the form of UML 2.0 state machines. State machines combine three properties that make them very useful in practical service engineering; they enable readable and precise definitions of stateful behavior; the behavior can be analyzed formally using model checkers and other tools; and it can be effectively implemented in

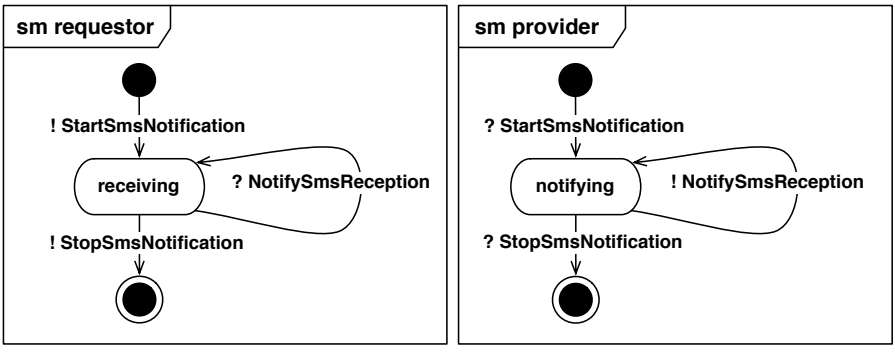


Fig. 7. State machines for active SmsNotification service role behaviors

a variety of ways to satisfy non-functional requirements such as performance. In short, state machines provide a working basis for model driven development[3]. This combination of properties is the main reason why we use state machines in service component design. (In many cases it is useful to combine state machines with other formalisms such as temporal logic, Petri nets and process algebras, but this will not be elaborated here).

When it comes to implementation, there are many issues to consider. We have found that most can be addressed quite systematically as explained in [20], and hidden from the application/service developer. Many people believe that asynchronous messaging will incur performance penalties compared to synchronous method invocation. This is true for local invocations within one address space and thread, but in a distributed system with multithreading the opposite is normally the case. Our execution model relies on asynchronous communication between components using signal buffers. This ensures a decoupled component model well suited for modeling and realization of distributed systems. See [9] for a detailed look on our execution model.

5.4 Loose Coupling and Contracts

Loose coupling between components is a commonly expressed goal when constructing a service-oriented architecture. A key to achieve loose coupling is to minimize the knowledge that needs to be shared between communicating partners. Organizing functionality in services described by interfaces is a considerable simplification, but the reduction is too extensive if only the static invocation details are described. In addition to the list of operations, there usually exists an intended invocation order. The tenets cited in Section 2.1 list contract and schema as the shared service knowledge in service-orientation, and our collaboration oriented modeling concept encompasses this through interfaces and role state machines. The signals listed as signal receptions in the interfaces of the service roles provide the data structures that form the schema of a service. Likewise, the service role state machines for the requestor and provider roles

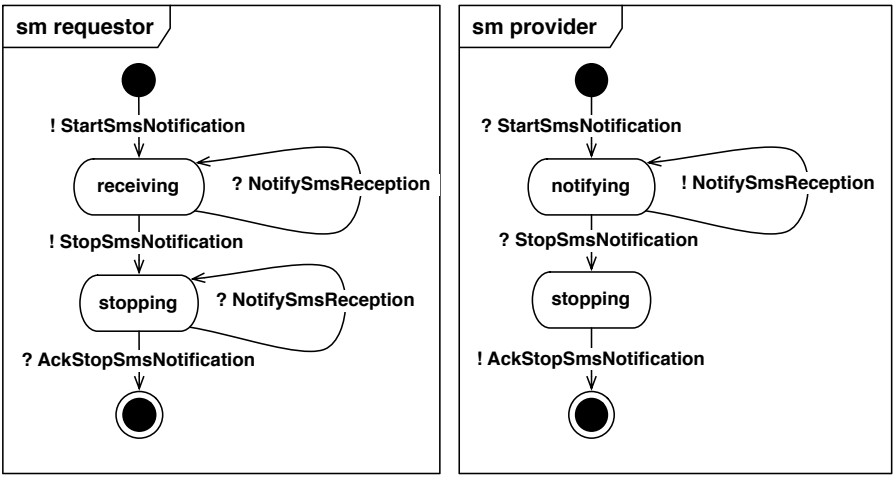


Fig. 8. Corrected state machines for active SmsNotification service role behaviors

together define a behavioral contract that entities requesting and/or providing the service must adhere to. This contract precisely describes the behavioral semantics of a service, in other words the expected and accepted behavior of the involved entities while omitting other more component-specific details.

6 Related Work

Treating service as a collaborative activity is not a new idea. For instance, CoSDL[19,18] is a collaborative approach for designing systems with SDL, defining collaboration modules that encapsulate the required interactions of all agents in a collaboration. Our use of the Collaboration element encapsulates these using an established UML 2 concept together with specialized stereotypes suited for our service model.

The use of roles in modeling was fundamental in the OOram[17] methodology, and later influenced OMG in its work on collaborations in UML.

COSMO[16] is a proposed conceptual framework for service modeling and refinement, aimed at providing a common semantic meta-model for service languages. COSMO identifies two roles involved in a service; the user role and provider role, and our requestor and provider roles correspond well to these concepts. It seems that COSMO does not treat initiative, so whether initiative is restricted to the user role alone is not clear. The examples in [16] are all with client-server type initiatives, while our service concept are deliberately devised to allow peer-to-peer style initiatives. Causality relations between activities is the notion of state in COSMO, while we have chosen to deal with state explicitly with state machines when modeling the roles.

The notion that services can be of active nature is taken for granted in Vissers and Logrippo's classic paper about the service concept [24], which define a service as "the behavior of the provider as observed by its users". The provider behavior is seen as an abstract machine which service primitives can be invoked upon, but it is also capable of spontaneous actions resulting in the execution of service primitives upon the user. Our use of state machines to describe service role behavior somewhat resembles this, but we also describe the interaction protocol of the service, which corresponds to the "internals of the service provider" in [24].

Our ServiceRoleStateMachine is very similar to the Port State Machine proposed by Mencl [11]. But while the Port State Machine is a thorough extension of the Protocol State Machine in UML 2, our ServiceRoleStateMachine is just a very restricted Behavioral State Machine, constructed solely for use in our service modeling concept. The strength of the ServiceRoleStateMachine is its simplicity, which makes it easier to validate, while the Port State Machine is a versatile construction suitable for general modeling using UML.

7 Summary

The service concept of contemporary SOA is of passive nature, while functionality involving active participants are of an active nature. We have introduced the notion of active services that include active participants, by allowing both the requestor and provider to take communicative initiatives.

We have applied the UML 2 Collaboration concept that provides a modeling context well suited for service specification, defining a boundary at the modeling level. Service interactions are modeled by state machines communicating using asynchronous signals. This emphasizes the boundary crossings related to the physical distribution. It also provides loose coupling and execution independence of the entities involved in the service. The service autonomy is strengthened by enabling both requestor and provider roles of a service to take initiative. There is no single controlling entity in an active service, just as there is no such entity in a SOA as a whole. The state machines of the requestor and provider roles establish a precise behavioral contract for the service. The signal receptions of the roles together with the data structures of the signals, form the schemas that are shared in a service. We are not concerned with the exact type that eventually will fill a role in the service, as long as it is compatible with the schema and contract effectively specified by the service collaboration. We do not treat semantic compatibility based on policies in this paper, but the work of Sanders [22,21] addresses substitutability of service roles by the use of goal expressions. These expressions are attached to a collaboration as a whole, as well as to the states of the roles, and could be applied to the service collaboration described here in order to address the semantic comparability of service instances.

Contemporary SOAs does no treat states explicitly, and we contend that stateful behavior should be dealt with as explicitly as possible. As demonstrated by our SMS example in section 5.2, even small and uncomplicated functionality is

likely to suffer from mistakes. We have separated the service modeling from the implementation, and are able to validate the asserted behavior since the service interactions are formalized as communicating state machines.

References

1. Box, D.: A guide to developing and running connected systems with indigo. MSDN Magazine 19(1) (January 2004)
2. Bræk, R.: Using roles with types and objects for service development. In: Yongchareon, T., Aagesen, F.A., Wuwongse, V. (eds.) SMARTNET. IFIP Conference Proceedings, vol. 160, pp. 265–278. Kluwer, Dordrecht (1999)
3. Bræk, R., Melby, G.: Model-Driven Service Engineering. In: Model-Driven Software Development. Part III, pp. 385–401. Springer, Heidelberg (2005)
4. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR, Upper Saddle River (2005)
5. Erl, T.: A W3C Web Services Glossary (March 2007), `http://www.ws-standards.com/glossary.asp`
6. Floch, J., Bræk, R.: ICT convergence: Modeling issues. In: Amyot, D., Williams, A.W. (eds.) SAM 2004. LNCS, vol. 3319, pp. 237–256. Springer, Heidelberg (2005)
7. Parlay Group. Parlay X Web Services Specification, Version 2.1 - Short Messaging (2006), `http://www.parlay.org/en/specifications/pxws.asp`
8. Holzmann, G.J.: The SPIN model checker: Primer and reference manual. Addison-Wesley, Reading (2004)
9. Kraemer, F.A., Herrmann, P., Bræk, R.: Aligning UML 2.0 state machines and temporal logic for the efficient execution of services. In: Meersman, R., Tari, Z. (eds.) OTM 2006. LNCS, vol. 4276, pp. 1613–1632. Springer, Heidelberg (2006)
10. Kraemer, F.A., Samset, H.: Ramses User Guide. Avantel Technical Report 1/2006. Technical report, Department of Telematics, NTNU, Trondheim, Norway (2006)
11. Mencl, V.: Specifying component behavior with port state machines. Electronic Notes in Theoretical Computer Science 101C, 129–153 (2004); In: de Boer, F., Bonsangue, M. (eds.) Proceedings of the Workshop on the Compositional Verification of UML Models CVUML
12. Moreau, J.-J., Weerawarana, S., Ryman, A., Chinnici, R.: Web services description language (WSDL) version 2.0 part 1: Core language. W3C recommendation, W3C (June 2007), `http://www.w3.org/TR/2007/REC-wsdl20-20070626`
13. Natis, Y., Schulte, R.: Advanced SOA for advanced enterprise projects. Technical report, Gartner Group (2006)
14. OASIS. Reference Model for Service Oriented Architecture v1.0 (October 2006)
15. Object Management Group. Unified Modeling Language 2.0 Superstructure Specification (2006)
16. Quartel, D.A.C., Steen, M.W.A., Pokraev, S., van Sinderen, M.: COSMO: A conceptual framework for service modelling and refinement. Information Systems Frontiers 9(2-3), 225–244 (2007)
17. Reenskaug, T., Wold, P., Lehne, O.A.: Working with Objects: The OOram Software Engineering Method. Prentice-Hall, Englewood Cliffs (1995)
18. Rößler, F., Geppert, B., Gotzhein, R.: Collaboration-based design of SDL systems. In: Reed, R., Reed, J. (eds.) SDL 2001. LNCS, vol. 2078, pp. 72–89. Springer, Heidelberg (2001)

19. Rößler, F., Geppert, B., Gotzhein, R.: Cosdl: An experimental language for collaboration specification. In: Sherratt, E. (ed.) SAM 2002. LNCS, vol. 2599, pp. 1–20. Springer, Heidelberg (2003)
20. Sanders, R.T.: Implementing from SDL. Telektronikk 96(4) (2000) ISSN 0085-7130
21. Sanders, R.T.: Collaborations, Semantic Interfaces and Service Goals - a new way forward for Service Engineering. PhD thesis, Norwegian University of Science and Technology (NTNU) (2007)
22. Sanders, R.T., Bræk, R., Bochmann, G., Amyot, D.: Service discovery and component reuse with semantic interfaces. In: 12th SDL Forum, Grimstad, Norway (2005)
23. Sanders, R.T., Castejón, H.N., Kraemer, F.A., Bræk, R.: Using UML 2.0 collaborations for compositional service specification. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 460–475. Springer, Heidelberg (2005)
24. Vissers, C.A., Logrippo, L.: The importance of the service concept in the design of data communications protocols. In: Diaz, M. (ed.) PSTV, pp. 3–17. North-Holland, Amsterdam (1985)