DYNAMIC AND DECENTRALIZED SERVICE COMPOSITION

With Contextual Aspect-Sensitive Services

Thomas Cottenier, Tzilla Elrad

Computer Science Department, Illinois Institute of Technology, 3300 S. Federal Street, Chicago, USA

Keywords: Web Services and Web Engineering, Service Composition, Context-Sensitive Customization.

Abstract: This paper introduces a new technique to dynamically compose Web Services in a decentralized manner. Many of the shortcomings of current Web Service composition mechanisms stem from the difficulty of defining, modularizing and managing non-functional concerns and context-sensitive behaviours. Contextual Aspect-Sensitive Services (CASS) is a distributed aspect platform that targets the encapsulation of coordination, activity lifecycle and context propagation concerns in service-oriented environments. CASS enables crosscutting and context-sensitive concerns to be factored out of the service implementations and modularized into separate units of encapsulation. CASS does not require a centralized orchestration engine to coordinate the message exchanges. Coordination logic is woven directly at the level of the message processing engine. The CASS composition definition language offers a powerful alternative to static and centralized business process definition languages such as BPEL4WS.

1 INTRODUCTION

Service-oriented, peer-to-peer and pervasive computing environments are pining for dynamic composition mechanisms.

In service-oriented environments, distributed components are discovered and integrated at runtime. Consequently, they can hardly anticipate the functionality requirements of all their client applications. There is therefore a need for methods to customize service components based on context specific requirements.

Service composition is a way to generate custom behaviour. Nevertheless, current composition mechanisms, such as orchestration languages tend to produce highly specialized compositions that are particular to a specific situation. They do not accommodate changing requirements very well and target long term collaborations. Some service collaborations only make sense in a particular context, and have a short life cycle. Hence, there is a need for mechanisms that support the dynamic deployment and refinement of service compositions. Therefore, non-functional concerns and context-sensitive behaviours imperatively need to be factored out of the core logic of collaborations. Those concerns tend to be hard to modularize into regular components, and are therefore good candidates for aspect-oriented modularization.

The Contextual Aspect Sensitive Service (CASS) platform is a distributed aspect platform that targets the encapsulation of coordination, activity lifecycle and context propagation concerns in service-oriented environments.

1.1 Service-Oriented Architecture

Service-Oriented Architecture (SOA) is an architectural style whose goal is to achieve loose coupling among interacting software entities, in order to facilitate the integration of software components into distributed applications. Loose coupling is obtained by limiting the complexity of the service interfaces, which should only encode generic semantics. Therefore, all application-specific semantics have to be encoded in descriptive messages. Second, in order to ensure interoperability, the messages have to be written in an open standard format that is understood by all parties, generally XML. Moreover, the message structure should be extensible to guarantee evolvability. Finally, an SOA must provide a discovery mechanism that enables services to be discovered, bound and interactively accessed.

Cottenier T. and Elrad T. (2005).
DYNAMIC AND DECENTRALIZED SERVICE COMPOSITION - With Contextual Aspect-Sensitive Services.
In *Proceedings of the First International Conference on Web Information Systems and Technologies*, pages 56-63
DOI: 10.5220/0001232000560063
Copyright © SciTePress

1.2 Web Services

Web Services is a collection of SOA implementation technologies based on open standards and interfaces, including XML, SOAP, WSDL and UDDI (Walsh, A., 2002). It is generally accepted that Web Services are an SOA implementation that exchange XML messages over an internet based transport protocol such as HTTP.

SOAP is a lightweight XML-based information exchange protocol for Web Services. A SOAP message envelope describes what is in a message and how to process it. The SOAP encoding defines a set of rules for mapping programmatic types to XML. WSDL is an XML-based language for defining Web Services interfaces that describe them and specify how to access them. UDDI is an XML-based protocol for registering Web service descriptions, discover and retrieve them.

A service provides a handle as an URI. A service implementation is composed of one or more operation providers. The interfaces of those providers are called port types and are defined in the service WSDL. A port type defines a set of operations that are supported by the operation provider, as well as the structure and types of the input and output messages that are supported by these operations.

2 SERVICE COMPOSITION

Web services aim at being building blocks for applications. A *business process* is composed from multiple component services given a process definition. Service composition mechanisms need to maintain state across components and manage activity contexts, exceptions, and transactional integrity.

2.1 Orchestration and Choreography

Service composition can be seen from two main viewpoints: orchestration and choreography. The choreography viewpoint covers the perspective of a collaboration between several services that realizes a value chain. It describes the interactions between service providers. The orchestration viewpoint describes the behaviour that a service provider performs internally within a collaboration.

Choreography languages are not mature yet. Only the orchestration perspective has been widely adopted in the industry, through the adoption of the Business Process Execution Language for Web Services specification (BPEL4WS, 2003). There is a therefore a tendency to equate web service composition with service orchestration although languages such as BPEL only provide a client-centric perspective of service composition. BPEL is not suited for expressing any kind of choreography.

Furthermore, some service collaborations only make sense in a particular context, and have a short life cycle. There is therefore a need for mechanisms that support dynamic deployment and refinement of short term collaboration. Orchestration languages only target static web service collaborations.

Finally, the hierarchical modularization of the composition definition does not allow the encapsulation of some aspects of the orchestration such as exception handling or authentication (Charfi, A, 2004), business rules (Verheecke, B, 2004) or activity management.

For instance, consider how activities are implemented. The activity concept provides a way to scope arbitrary units of distributed work. Fig. 1 illustrates an activity that involves 4 web services. The work of the participants in the activity is correlated by propagating some context information that is maintained by a context service. An activity life-cycle service further enhances the context information and informs the participants about the lifetime of the activity. Neither the choreography nor the orchestration perspective cleanly modularizes context propagation and lifecycle concerns. Context propagation and activity lifecycle code is scattered and tangled with the core functionality of the activity.

Tangled code is hard to understand, impede code reusability and may leave a negative impact on adaptability, system performance, and reliability.

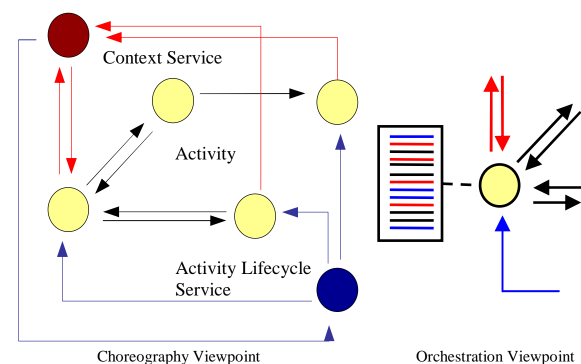

Figure 1: Current choreography and orchestration definition languages do not support the modularization of context-propagation and activity lifecycle concerns. Context propagation and activity lifecycle code is scattered and tangled with the core functionality of the activity.

2.2 Aspect-Oriented Service Composition

This paper proposes an aspect-oriented service composition approach based on collaboration-based design. Aspect-Oriented Software Development (AOSD) (Kiczales, G, 1997) is an extension to other software development paradigms that allows capturing and modularizing concerns that crosscut a software system into modules called aspects. Aspect-Oriented Programming (AOP) makes very powerful program transformations possible, through a composition process where pieces of code called *advices* are *woven* into the core program at locations called *pointcuts.* A *joinpoint* is an instance of an event intercepted by a pointcut expression. Members and methods can also be inserted in classes through an aspect construct called *inter-type declaration.*

Aspects have the ability to introduce functionality in a core program in a non-invasive way, making it possible to alter the behavior of a system a posteriori. Aspect weaving can be done either at compile time, load time, or even at runtime.

The CASS service refinement approach builds on top of Collaboration-Based Design (VanHilst, M., 1996, Smaragdakis, Y, 2002) techniques applied to the development of distributed systems. Aside class, component or service-based decomposition, collaboration-based design advocates decomposing applications into a set of collaboration layers.

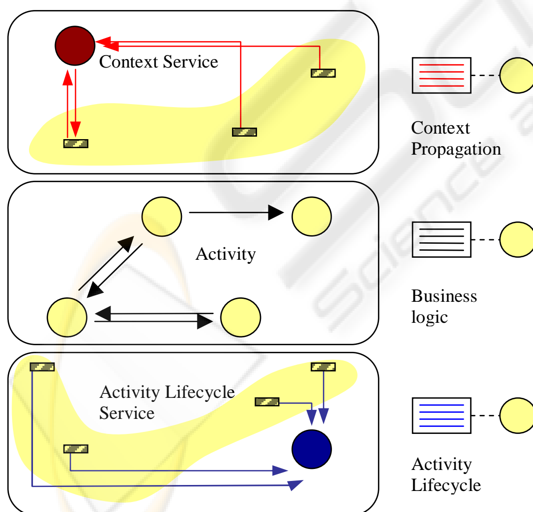

Figure 2: Decomposition of an activity into collaboration layers. Context-propagation and life-cycle concerns are encapsulated in distinct collaboration layers. Both the orchestration and the choreography perspectives of a collaboration layer are modular.

A collaboration layer defines how each actor contributes to a given task or feature. It captures the protocols services should implement to fulfill an interaction. A role specifies the responsibilities the core service should take up in order to take part in the interaction. If the collaborations are fairly independent, systems can be built incrementally by composing independently defined collaborations layers.

CASS factors out the various concerns that arise when services are combined into distinct collaboration layers. A distributed pointcut language is used to recompose them non-invasively. The CASS aspect constructs are powerful enough to define service choreographies. CASS supports a set of pointcut and advice composition operators that allows the control flow and the data flow of a business process to be defined in a concise way.

3 CONTEXTUAL ASPECT-SENSITIVE SERVICES

3.1 Requirements

CASS targets the following requirements:

a. *Encapsulation of context-sensitive collective behavior.* The most variable elements are not the services themselves, but their interactions. There is a need for reusable units of encapsulation that have the ability to capture complex interactions that cut across services and heterogeneous containers.

b. *Non-invasive refinement.* The refinement of a client/server relationship needs to be non-invasive. It cannot require changes to the implementation of the client or service entities.

c. *Concurrent customization.* The same service instance might need to simultaneously offer customized services to other entities, adapted to their profile and their context.

d. *Loose-coupling.* CASS collaboration layers may not break the loose coupling requirements of service-oriented architectures and need to accommodate platform heterogeneity.

3.2 CASS Platform

CASS intercepts SOAP messages at the boundaries of service components, both at the client and the service side. CASS includes native mechanisms to transform, compose, synchronize and multicast the intercepted messages.

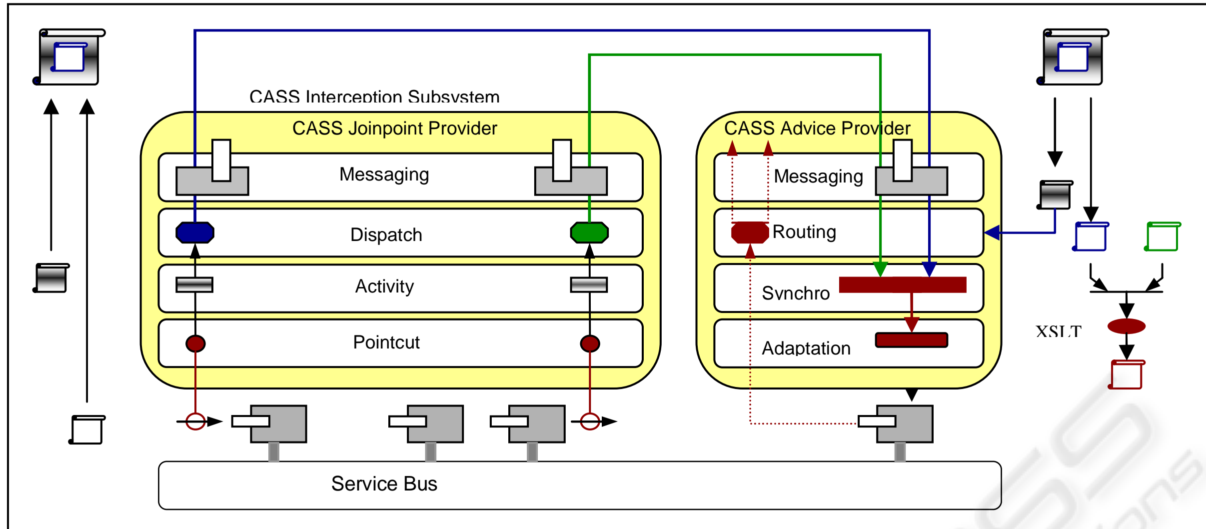

Figure 3: Message interception components

The interception subsystem is composed of 6 distinct components: an interceptor component, an activity component, a dispatcher component, a composition component and an adaptation component. CASS also includes an asynchronous messaging subsystem. Fig.3. illustrates how those components interact, for messages that flow from their interceptors to a remote service method.

The interceptor component is responsible for filtering the messages that flow in and out the container. If the message matches a pointcut expression, it is intercepted and its processing is interrupted.

The activity component attributes an interaction context to the intercepted message. This context information is used to delimitate activities. Messages that share a common context are part of the same activity.

The dispatcher component is responsible for forwarding the intercepted messages to remote service method. The message can be sent to multiple services in parallel.

The synchronization component can delay the processing of an intercepted message until the occurrence of another event in the life-cycle of the activity in which the message interaction takes part. The synchronization component buffers the intercepted messages and coordinates the message flow within an activity.

The adaptation component can transform both the structure and the content of the intercepted messages, before they are dispatched to the service method.

The communication mechanism used to forward the intercepted message to a remote service method is independent of the type of interface exposed by the method. The messaging component can transparently replace an RPC-based interaction by a pair of asynchronous request/response messages. The communication mechanism used is specified at the level of the pointcut definition. When the messaging mechanism is used, the routing component is responsible for directing the service response back to the joinpoint.

The CASS components interpret declarative specifications. A rich set of composition and synchronization operators enable the control flow and the data flow of choreographies to be specified in a concise way.

4 MESSAGE INTERCEPTION AND COMPOSITION

4.1 AXIS

Apache eXtensible Interaction System (Axis) is an implementation of the Simple Object Access Protocol (SOAP). The Axis engine processes incoming and outgoing messages by invoking a chain of handlers that manipulate an object called message context. The message context object is a structure that contains a request message, a response message and a set of properties. Global message handlers perform some SOAP specific processing such as serialization, message dispatching or routing. Service-specific message handlers perform tasks that enforce the non-functional properties of the messages that flow through the container such as authentication, encryption or session and persistency management.

4.2 Message Interception

CASS uses an aspect-oriented programming language to intercept the message context at different points of the handler chain. When a message matches a regular expression defined in a pointcut expression, message processing is interrupted, and the CASS interception subsystem takes control over the message.

An interceptor can be credited some additional properties that depend on the message handler it acts upon. Fig 4 shows how different qualities of interceptors are specified using aspect pointcut expressions that act up different message handlers.

Quality enforcement rules specify constraints on the interceptors that guarantee essential invariants of the system are not violated. For example, a quality rule might require that an interceptor that catches a decrypted message exhibits appropriate security credentials.

CASS service-oriented pointcut expressions specify a set of message interceptors that operate within a collaboration layer. The CASS pointcut interpreter translates a declarative pointcut expression that is defined with respect of WSDL definitions into a lower level aspect pointcut expression that acts upon the message handlers.

The code sample of Fig 5 defines a pointcut on calls to operations of the “MathPortType” of the “MathService” on the IIT application server.

```
<pointcut  name = "MathPointcut"
       type = "client"
       service = "*/MathService"
       operation = "* AND !multiply"
       host = "http://www.iit.edu:8081"/>
```

Figure 5: CASS client-side pointcut expression

4.3 Message Dispatching

When a message matches a pointcut expression, the execution of the message handler on which the pointcut acts is interrupted and the message context being processed is intercepted.

The dispatching component can perform 3 different actions on the message context:

a. *Before forwarding*. The intercepted message context is forwarded to a remote service method before it is processed by the message handler. The remote method can modify the input message of the message context. The interrupted handler always resumes.
b. *After forwarding*. The intercepted message context is forwarded to a remote service method after the message handler returns control – after the interrupted service method has been executed. The remote method can modify the output message of the message context.
c. *Around execution.* A remote service method is executed instead of the message handler. Both the input and the output messages can be modified by the remote service method. The remote method can resume the execution of the interrupted handler, by invoking a special method, ‘proceed’. The ‘proceed’ method has the same signature as the remote method itself. If the ‘proceed’ method is not called within the remote service method, the initial service call or execution is not executed at all.

The action to be performed on an intercepted message is specified in an advice definition. The advice definition defines the type of action and the host, service and operation to which the intercepted message must be dispatched. Advice definition can be composed and bound to multiple pointcut expressions.

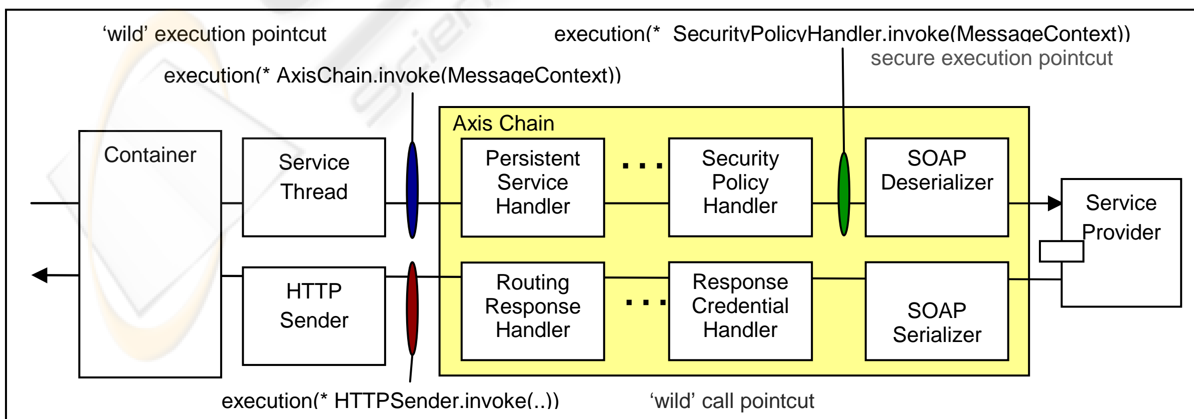

Figure 4: Message Context interceptors – CASS pointcut expressions are translated into lower level pointcut expressions that act upon the message handlers. Depending on the handler that is intercepted, CASS pointcuts can be attributed quality properties.

```
<pointcut name= "redir" type= "client"
      service= "MathService"
      operation= "add"
      host= "http://www.iit.edu:8081"/>
<advice type= "around" bind-to= "redir"
      service= "MathService"
      operation= "add"
      host="http://ws-concerns.com:8081"/>
```

Figure 6: Redirection aspect

The code sample of Fig 6 illustrates a simple redirection aspect. A call to 'MathService' on the IIT server is replaced by a call to the same service on an alternate server. Such aspects can be used to implement load-balancing and fault tolerance algorithms.

4.4 Message Adaptation

The intercepted messages may very well not match the remote service method interface. The adaptation component can transform both the structure and the content of the intercepted messages, before they are dispatched to the remote service method. The adaptation component uses XSLT to generate a new SOAP message that conforms to the interface of the new method interface. The XSL templates used for the transformation are part of the CASS composition specification. The adaptation component makes it possible for remote service methods to process a wide range of intercepted messages.

4.5 Context Propagation

A context uniquely defines the scope of an activity. Contexts are transparently propagated along interactions of a same activity. Contexts are declared at the pointcut level. Each time an event triggers a contextual pointcut, the intercepted message is wrapped into a CASS envelope that includes a context description. The context description carries the following information:

a. *Context name*. The context name is the name of the pointcut that defines the context
b. *Context id*. A different id is attributed to each joinpoint instance.
c. *Joinpoint callback*. The URI of the callback interface of the joinpoint that created the context instance.
d. *Context service data*. Context data associated to the activity. The context data can also be maintained as service data associated to a service. In the later case, only a URI to the service data is propagated.

The context id discriminates different activity instances. Pointcuts can therefore be defined on a per-activity basis. Contextual pointcuts enable concurrent service customization.

4.6 Control Flow Constructs

The CASS specification can be used to define service choreographies. Calls and executions of remote service methods can themselves be intercepted, and bound to other service methods. CASS introduces a set of pointcut and advice composition operators that allow business processes to be defined.

4.6.1 Advice Chain

A sequence of advice methods can be bound to a pointcut. The code sample of Fig 7 shows how a chain of advices is specified. The sequence defines a stack of methods to be executed around the 'MathPointcut'. The advice methods are deployed on different hosts. Fig 8 illustrates the message flow of the advice chain.

First, the intercepted message is forwarded to the filter1 operation of the advice1 service. The request message is XSLT transformed, so it matches the signature of the filter1 operation. Advice1 processes the request message, and calls its 'proceed' method. The message is mapped and forwarded to the next method on the advice stack, filter2. filter2 further processes the request message and calls 'proceed'. When no more method is left on the advice stack, control is returned to the joinpoint. The request message is mapped back from filter2 to filter1, then from filter1 to the operation interrupted by the pointcut. The interrupted operation proceeds and operates on the mapped request message returned by filter2. Once it completes, the response message is returned back to the last method on the advice stack – in this case, filter2. filter2 processes the response message, returns it to filter1, which returns control to the joinpoint.

The interaction between the joinpoint and the advice methods is really peer-to-peer. There is no center of coordination that controls the message flow. Such collaboration can not be specified using an orchestration language as BPEL.

4.6.2 Concurrency and Synchronization

Several advice methods bound to the same interceptor can be executed concurrently, by composing them with the 'flow' operator. Flow composition has multicast semantics.

```
<sequence bind-to="MathPointcut">
  <advice type="around"
         service="MathAdvice1" operation="filter1"
         host="http://www.iit.edu:8080">
   <request>
     <xsl:template – message mapping from pointcut
                     request to advice 1 request/>
     <xsl:template - message mapping from advice 1
                     request to pointcut request/>
   </request>
   <response>
     <xsl:template - message mapping from pointcut
                   response to advice 1 response/>
     <xsl:template - message mapping from advice 1
                   response to pointcut response/>
   </response>
  </advice>
  <advice type="around"
         service="MathAdvice2" operation="filter2"
         host="http://ws-concerns:8080">
   <request>
    <xsl:template  - message mapping from advice 1
                     request to advice 2 request/>
    <xsl:template  - message mapping from advice 2
                     request to advice 1 request/>
   </request>
   <response>
     <xsl:template - message mapping from advice 2
                   response to advice 1 response/>
     <xsl:template - message mapping from advice 1
                   response to advice 2 response/>
   </response>
  </advice></sequence>
```

Figure 7: Advice sequence. 2 'around' advices are sequentially bound to a single pointcut expression. XSL templates adapt the request and response messages so they match the interface of the service they are redirected to.

Control can be returned to the joinpoint in 2 ways. In the first case, control is returned when the joinpoint callback listener receives its first response message. The other responses are ignored. This construct is handy for querying semantically equivalent services concurrently, or when no reply is expected from the advice methods.

In the second case, the joinpoint synchronizes the responses. All the response messages are buffered and control is only returned to the joinpoint when all calls have returned. The buffered messages are then processed by the XSLT-based adapter, which aggregates the results. Powerful operations on the content of response messages can be specified at this level. For example, the XSLT mathematical functions make it possible to compute the sum, the product or the minimum value of response messages variables.

Pointcuts can also be correlated by composing them with the 'join' operator. 'join' synchronizes interceptors acting on different services. Processing is interrupted until all the composed pointcuts have been triggered. All the intercepted messages are forwarded to the advice method where they are buffered. Once all the pointcuts have been triggered, the XSLT adapter aggregates the messages into one request that is processed by the advice.

The combination of the 'flow' advice composition operator and the 'join' pointcut composition operator provide an elegant way to implement Petri-Net style flows, such as fork/join operations on different hosts. BPEL does not support this kind of compositions.

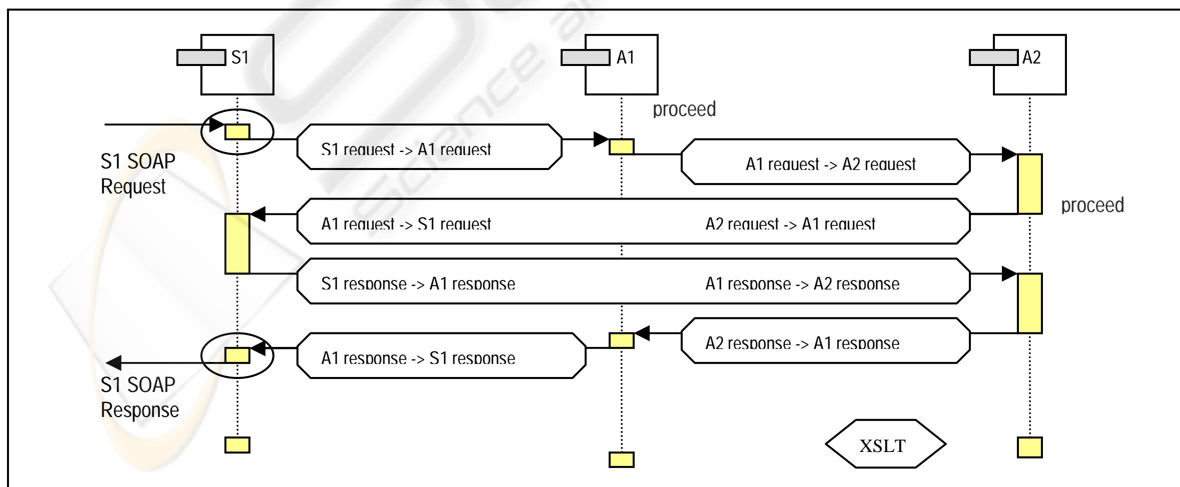

Figure 8: Sequence diagram for a chain of 2 'around' advices. Mappings need to be specified in both direction, in case the 'proceed' method is called by the advice methods. The interaction between S1, A1 and A2 is truly peer-to-peer. There is no center of coordination in this interaction

5 DYNAMIC DEPLOYMENT

The CASS platform includes a dynamic aspect deployment service. A client parses the choreography specification, and partitions it into pointcut and advice definitions that are relevant to each participant host and service. The deployment service translates those descriptors into low-level pointcut expressions that act upon the message handlers.

All pointcuts are associated to an activity context. Pointcuts are only triggered for messages that match the pointcut expression, and whose context descriptor corresponds to the pointcut activity. The activity context guarantees consistent deployment of choreographies. The pointcut that defines the entry point of the activity – the context – is only activated after successful deployment of the choreography. This ensures that the messages being processed at deployment time are not affected until all interceptors, advice handlers and adapters are deployed and activated.

The CASS choreography definition is an intermediate representation. It is typically generated from higher level definitions and automatically validated. CASS Interaction patterns are encoded into XSLT files, where all service specific information is factored out, and represented by XSL variables. A graphical tool can then used to map the interaction patterns to concrete service topologies.

6 CONCLUSION

This paper introduces a new technique to dynamically compose Web Services in a decentralized manner. Contextual Aspect-Sensitive Services (CASS) enables crosscutting and context dependent behaviour to be factored out of the service implementations and modularized into separate units of encapsulation that are exposed as Web Services.

CASS introduces a service-oriented pointcut model and composition operators. The CASS platform implements a context propagation mechanism that transparently maintains activity context across service collaborations in a distributed setting. The context information is used to define sophisticated pointcuts that enable concurrent service customization.

CASS can be used to dynamically deploy services choreographies. It has the potential to define more advanced collaboration scenarios than can be specified with state-of-the-art web service orchestration languages such as BPEL4WS.

ACKNOWLEDGEMENT

This work is partially supported by CISE NSF grant No. 0137743.

REFERENCES

Walsh, A., 2002. UDDI, SOAP, and WSDL: The Web Services Specification Reference Book, Prentice Hall.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., Irwin, J., 1997. Aspect-oriented programming. In Proceedings of the European Conference on Object-Oriented Programming. Springer-Verlag.

Filman, R., Friedman, D., 2000. Aspect-oriented Programming is Quantification and Obliviousness. In Workshop on Advanced Separation of Concerns, OOPSLA 2000.

VanHilst, M., Notkin, D., 1996. Using Role Components to Implement Collaboration-Based Designs. In Proceedings of the 11th ACM conference on Object-Oriented Programming, Systems, Languages, and Applications.

Smaragdakis, Y., Batory , D., 2002. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. ACM Transactions on Software Engineering and Methodologies

Chafle, G., Chandra, S., Mann, V., Nanda, M. G., 2004. Decentralized Orchestration of Composite Web Services. In Proceedings of the Thirteenth International World Wide Web Conference.

Charfi, A., Mezini M., 2004. Aspect-Oriented Web Service Composition with AO4BPEL. In Proceedings of European Conference on Web Services.

Verheecke, B., Cibrán, M. A., Jonckers, V. 2004., Aspect-Oriented Programming for Dynamic Web Service Monitoring and Selection, In Proceedings of the European Conference on Web Services.

Suvee , D. Vanderperren, W., Jonckers, V., 2003. JAsCo: an aspect-oriented approach tailored for component based software development. In Proceedings of the 2nd International Conference on Aspect-Oriented Software Development.

Cottenier, T., Elrad, T., 2004. Validation of Aspect-Oriented Adaptations to Components. Ninth Internationapl Workshop on Component-Oriented Programming as part of ECOOP'04

BPEL4WS, 2003. Business Process Execution Language for Web Services Specification (BPEL4WS) http://www-128.ibm.com/developerworks/library/ws-bpel

Axis, 2000. Apache http://ws.apache.org/axis