Contextual Pointcut Expressions for Dynamic Service Customization

Thomas Cottenier
Illinois Institute of Technology
cotttho@iit.edu

Tzilla Elrad
Illinois Institute of Technology
elrad@iit.edu

ABSTRACT

In service-oriented environments, components are discovered and integrated at runtime. The type of the client entities that expose potentially interesting contexts can therefore not be anticipated, and can not be subject to type-based pointcut expressions.

This paper proposes a contextual pointcut construct that attempts to address context passing mechanisms needed for concurrent customization of services.

Contextual pointcut expressions generalize the semantics of 'cflow' to enable advices to retrieve a richer set of context information along the call path to a target joinpoint. A context visitor collects information about the particular circumstances in which a target joinpoint is executed, including references to the joinpoints encountered along the path.

Contextual advices have the power to alter the control flow of the aspect execution, and return control to the joinpoints that are still alive in the target joinpoint's call path.

1. INTRODUCTION

The thoughts expressed in this position paper draw from the authors' experiences in applying AOP concepts and languages constructs to the development of a distributed aspect platform for dynamic and distributed service composition and customization

The particularities of distributed environments, and service-oriented environments in particular, force distributed aspect platforms to take a different perspective on context passing pointcut expressions and joinpoint composition.

Remote pointcuts [1] allow us to locally specify events of interest occurring on remote hosts. When the pointcut is triggered, the metadata of the joinpoint is passed to the local host and the corresponding advice is executed locally. Context passing from the joinpoint to the remote advice is part of the remote pointcut semantics.

However, more advanced context passing scenarios are not straightforward to support in a distributed setting.

In AspectJ, context passing from a client down through the calls that lead to a target service can be implemented as shown in Figure 1.

This type of context passing addresses the concurrent customization of services according to the context of a client entity. Concurrent service customization [2] means that a same service instance can be customized in one execution context without impacting clients interacting with the service instance in other contexts.

However, in service-oriented environments, components are discovered and integrated at runtime. The type of the client entities that expose interesting contexts can therefore not be anticipated, and can not be subject to type-based pointcut expressions. Tough, the contexts that are of potential use for service customization are matched by:

```
cflow( call(* *(..)) && target(obj))
                && execution(* Service.f(..))
```

This pointcut expression is triggered at each method call encountered from the root to the call to the Service.f() method. However, only the last client context encountered is exposed by the joinpoint. Each client context is overridden by the next one.

To tackle context-dependent dynamic service composition and customization, we propose to generalize this construction, so that the context of all encountered entities on the path from the entity that initiates the interaction to the final caller is potentially available at the service side. We therefore attach a visitor to the cflow pointcut that accumulates the encountered context references instead of overriding them.

The paper is structured as follow. Section 2 briefly introduces Contextual Aspect-Sensitive Services and its context passing construct. Section 3 illustrates the need for a generalization of 'cflow' through a distributed advice chaining example. Section 4 introduces contextual pointcut expressions and section 5 discusses dynamic service customization. Finally, section 6 concludes this paper.

```
pointcut myServiceCall(Client client): cflow( call(* *(..)) && target(client) );
pointcut myServiceExec(Service service): execution(* Service.f(..)) && this(service);
pointcut myServiceContext(Client client, Service service): myMethodCall(client) &&
                                                             myMethodExec(service);
```

Fig.1. Caller context passing for a remote pointcut

2. SERVICE-ORIENTED POINTCUTS

The Contextual Aspect-Sensitive Service (CASS) [3,4,5] platform proposes a new technique to dynamically compose Web Services in a decentralized manner, by deploying SOAP message interceptors at the boundaries of Web Services. Remote pointcuts are declared with respect of the port types defined in the service WSDL definitions, to maintain platform independence.

CASS enables crosscutting and context-dependent behavior to be factored out of the Service implementations and modularized into separate units of encapsulation that are exposed as Web Services. Service orchestrations can then be defined in a much more flexible way, as services can be dynamically customized to address changing business rules or context-dependent requirements. Because CASS weaves the coordination logic directly at the level of web service interfaces, support for a 'perCflow' type of aspect construct is fundamental in order to provide the capability to specify the context under which a given orchestration should be activated.

CASS provides joinpoint and advice callback interfaces to deal with asynchronous message based interactions. These callback interfaces allows multiple advices bound to a shared joinpoint to be executed concurrently and also enable synchronization primitives to be embedded into pointcut expressions, using the 'join' pointcut composition operator. A pointcut that is composed of 'joined' pointcut expressions is triggered only once each one of the expression has been triggered in a specific context. Asynchronous joinpoints can handle not being returned control to, when their callback methods are never invoked.

In CASS, messages are intercepted at the joinpoints specified by a 'cflow' expression and wrapped into an envelope which contains the cflow expression, as well as the joinpoint instance reference. The system ensures the propagation of this information from caller to callee, and within a same service instance execution context.

The following specification shows how 'cflow' is implemented CASS. The effect of the 'cflow' aspect is to wrap calls to the 'MathService' web service into an envelope that contains the current execution context of the call.

The 'inContext' pointcut will only match the execution of an operation on 'MathService' if it is called in the context of the 'cflow.context' pointcut.

The smooth the progress of the discussion, a notation derived from AspectJ/AspectWerkz will be used in the following sections, instead of the Cass specification.

Next sections motivate the need for a generalization of 'cflow' that enables advices to retrieve a richer set of context information along the call path to a target joinpoint.

3. DISTRIBUTED ADVICE SEQUENCES

When defining a chain of advice in a distributed environment, the advices of the chain might very well execute on different hosts. In order to minimize unnecessary use of network resources, it is desirable to support direct chaining from one advice host to another, instead of letting the remote joinpoint dispatch the advice calls.

The example depicted in Figure 3 represents a simple B2B choreography. A billing service and a credit service perform some processing before and after an order is processed on a supplier service. Both the billing service and the credit service can force the transaction to rollback.

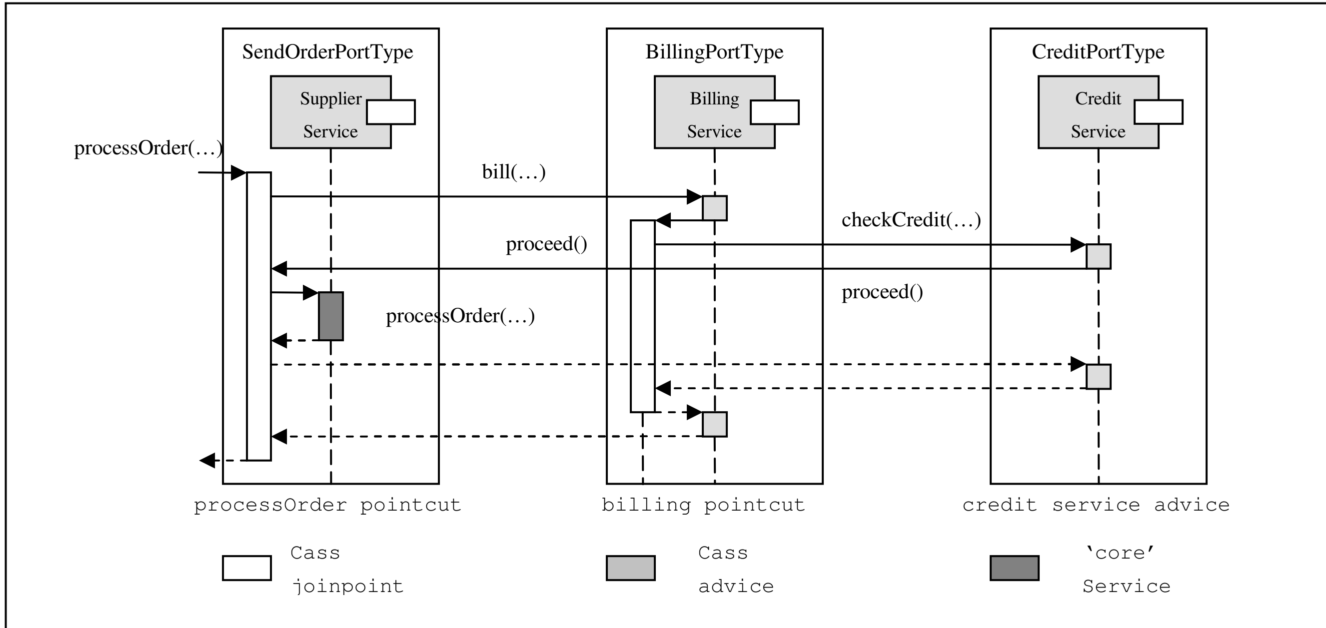
A chain of advices around the execution of the 'processOrder' method on the supplier service can be decomposed by declaring a pointcut expression on the call to the 'proceed' method of the joinpoint the 'BillingService' advice is bound to.

Instead of returning control to the billing joinpoint, the credit service advice returns control to the 'orderProcessing' joinpoint, that's higher on the call stack. This example illustrates how advices can return control to other joinpoints then the one that initially triggers them.

While this construct allows to effectively implementing chain of advices in a distributed setting, it forces the developer to pass along the root joinpoint reference from joinpoint to joinpoint, until the last advice.

```
<cass name="cflow">
   <pointcut name="context"
               service="edu/iit/concur/cass/testservices/MathService"
               operation="add"
               type="client"/>
   <advice name="contextPropagationAdvice"
               type="around"
               bind-to="context"
               host="http://localhost:8081"
               service="edu/iit/concur/cass/testservices/MathService"
               operation="add"/>
</cass>
<cass>
   <pointcut name="inContext"
               host="http://localhost:8081"
               service="edu/iit/concur/cass/testservices/MathService"
               operation="*"
               context="cflow.context"/>
</cass>
```

Fig.2. Cass specification of context pointcut




```
pointcut joinpoint_proceed(JoinPoint jp): call(* proceed()) && target(jp);
pointcut billing(JoinPoint jp): within(BillingService) && joinpoint_proceed(jp):
void around (JoinPoint jp) : billing(jp){
        checkCredit();
        jp.proceed();
        logTransaction();
}
```

Fig.3. Distributed sequence of advice on a shared joinpoint

This implementation is therefore very brittle and not reusable as the 'CreditService' advice implementation should be independent on whether it is used in an advice chain or not.

In order to pass the root joinpoint context to the advice chain, we propose to generalize the semantics of cflow to enable advices to retrieve a richer set of context information along the call path to a target joinpoint. Indeed, the joinpoints encountered in the chain are successively in the control flow of each other.

4. CONTEXTUAL POINTCUT EXPRESSIONS

Contextual pointcuts follow the same triggering rules as cflow pointcuts, but their joinpoints can accumulate context information during their life cycle. The context lives from the point the corresponding cflow expression evaluated to true until the system exits from the control flow.

Contextual pointcuts are defined by programmatically specifying a joinpoint context visitor. The joinpoint visitor is part of the pointcut definition rather than the advice definition.

We here adopt a notation that is closer to that of AspectWerkz. The arguments and return value of an advice are exposed by accessing the 'JoinPoint' parameter.

A visitor implements the 'Context' interface, which defines a 'visit' method and a 'proceed' method.

The 'visit' method is called by the runtime system each time a joinpoint is triggered in the control flow of the contextual pointcut the visitor is bound to.

The 'proceed' method allows an advice to return control, not only to the joinpoint that triggered it, but also to the other joinpoints encountered along that path.

The code sample of figure 4 defines a visitor for sequential chaining of advices in a distributed setting. On proceed, control is returned to the root joinpoint.

The advice sequence can now be defined as depicted in figure 5.

The visitor of figure 6 corresponds to the regular cflow behavior: each context is successively overridden by the next one.

```
pointcut processOrder = "execution(* SupplierService.processOrder(..))";
pointcut chain = "call(* Context.proceed()) && chainVisitor(processOrder)";

class chainVisitor implements Context{
        JoinPoint cjp;
        int i=0;
        public void visit(JoinPoint jp){
                if(i==0)
                        cjp=jp;
        }
        public Object proceed(){
                return cjp.proceed();
        }
}
```

Fig.4. Contextual Visitor for Distributed sequence of advice on a shared joinpoint

```
@Around("processOrder && chain")
public void bill(Context context){
        checkBillingInfo(…);
        context.proceed();
        doBilling(…);
}

@Around("within(BillingService) && chain")
public void checkCredit(Context context){
        checkCredit();
        context.proceed();
        logTransaction();
}
```

Fig.5. Distributed sequence of advice with contextual pointcut.

```
class cflowVisitor implements Context{
        JoinPoint cjp;
        public void visit(JoinPoint jp){
                cjp = jp;
        }
        public Object proceed(){
                return cjp.proceed();
        }
}
```

Fig.6. Contextual Visitor implementation of cflow

5. DYNAMIC SERVICE CUSTOMIZATION

In many cases, nothing is known about the type of the client entities of a service. There is then not other alternative than to expose the entire interaction context:

```
pointcut exposecallpath: context(call(* *.*(..)));
pointcut computation: exposecallpath &&
                 execution(* S.do_computation(…));
```

This kind of context propagation is however not scalable, and requires the context consolidation logic to be woven into all entities that are potentially in the path from the client to the service. There is therefore a need for mechanisms to discriminate entities on the base of the context they may expose. As this decision needs to be taken before the logic is woven into the entities, it needs to be expressed at the pointcut level.

In service-oriented environments, services can expose additional information about themselves in the form of Service Data. Service Data is set of structured data that describes static and dynamic properties of web service instance.

Figure 7 presents an example of a contextual visitor that uses service data to discriminate service contexts.

In the example, the visitor logic needs to be woven only at the interface of services that expose the 'ReliabilityServiceDataElement'.

Properties of this nature should therefore somehow be expressed at the pointcut level. Semantic Web techniques might be a way to achieve this goal.

5. CONCLUSIONS

This position paper proposes to generalize the semantics of 'cflow' to enable advices to retrieve a richer set of context information along the call path to a target joinpoint.

Contextual visitors collect information about the particular execution circumstances of a target joinpoint, including references to the joinpoints encountered along the path.

```
class ComputationContext extends Context{
  List reliabilityData = new ArrayList();
  List joinPoints = new ArrayList();
  JoinPoint lastjp;
  public void visit(JoinPoint jp){
    ServiceData sd = jp.getThis().
                     getServiceData();
    if(sd.contains("ReliabilitySDE")){
      reliabilityData.
       add(sd.get("ReliabilitySDE"));
      joinPoints.add(jp);
      lastjp = jp;
    }
  }
  public Object proceed(){
    if(computeReliability() > 0.95)
       return lastjp.proceed();
       ((JoinPoint)jps.get(0)).signal(
               new ReliabilityException()
       );
    }
    private float computeReliability(){ …}
}
```

Fig.7. Discrimination of context based on service data

When control is returned to the target joinpoint, contextual advices have the power to alter the control flow of the aspect execution, and return control to other joinpoints that are still alive in the target joinpoint's call path.

REFERENCES

[1] M. Nishizawa, S. Chiba, M. Tatsubori, Remote Pointcuts – A language Construct for Distributed AOP, In *Proceedings of the 3rd International Conference on Aspect-oriented Software Development,* Lancaster, UK, March 2004

[2] Eddy Truyen, Dynamic and Context-Sensitive Composition in Distributed Systems, PhD Thesis, October 2004.

[3] T. Cottenier, T. Elrad. Validation of Aspect-Oriented Adaptations to Components, *Ninth International Workshop on Component-Oriented Programming as part of ECOOP'04*, Oslo, Norway, June 2004.

[4] T. Cottenier, T.Elrad. Layers of Collaboration Aspects for Pervasive Computing, in proceedings of the *5th Argentine Symposium in Software Engineering* (ASSE'2004), Cordoba, Argentine, September 2004

[5] T. Cottenier, T. Elrad. Contextual Aspect Sensitive Services www.iit.edu/~concur/asc