

# Executable Choreography Processes with Aspect-Sensitive Services

Thomas Cottenier, Tzilla Elrad

Computer Science Department, Illinois Institute of Technology, 3300 S. Federal Street  
60616 Chicago, Illinois, USA  
{cotttho, elrad}@iit.edu

**Abstract.** This paper presents an executable service choreography framework (ECF). Current choreography languages are specification languages. They are used at design time to define a mutual contract between services that are under the supervision of different domain controllers. Choreography contracts are established by specifying the observable sequence of messages that are exchanged between services. 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 on-demand deployment of peer-to-peer collaborations. The ECF proposes to express the global contracts between domains using business rules rather than application-specific messages sequences. Service collaborations can then evolve and be deployed in a more flexible way. ECF uses a distributed aspect platform to enable dynamic superimposition of collaboration activities. ECF also defines a relationship between the distributed composition units and global collaboration models, so that the correctness of ECF choreographies can be checked for using existing service composition modeling tools.

**Keywords.** Business Process Modeling, Web-service Composition, Autonomic and on-demand computing, Aspect-Oriented Software Development

## 1 Introduction

The Executable Choreography Framework (ECF) aims at increasing the expressiveness and flexibility of cross-domain service composition. Current approaches to service composition tend to sacrifice much expressiveness and flexibility for added control and safety. While safety is a concern that cannot be ignored, today's service composition approaches enforce unnecessary constraints on the process logic. Processes are made more sequential than they need to be [6].

A major challenge to service composition is enabling truly peer-to-peer cross-domain service collaborations, and deploying them in a competitive time frame, while ensuring the safety of the participants is not compromised. In practice, participant corporate entities are unwilling to adopt mechanisms that allow peer-to-peer service interactions to be deployed when needed.

Domain controllers do not want to execute control flow elements of processes that are initiated and steered by other entities. They want to fully control logic that executes on their domain. Consequently, corporate entities are also reluctant to delegate part of the control flow of their business process. Control delegation is very inflexible. Any change to the process implementation requires a new agreement between controllers to be reached, before the modification can be deployed.

As a result, centralized composition approaches are today's state of the art. A centralized composition engine has immediate control over all message exchanges; all messages transit through the engine.

Tough, distributed compositions are better performing than centralized compositions. Distributed compositions partition the data and control dependencies between the components into smaller components that execute at distributed locations. As demonstrated in [6], decentralized execution brings performance benefits because centralized engines cannot fully take advantage of parallelism.

Moreover, advanced service composition scenarios are decentralized by their nature. They do not have a center of control. Centralized composition approaches force the composition model to be tweaked, so that it fits a centralized view of service composition. Decentralized approaches enable the composition implementation to fit the conceptual model of the composition better.

The position of the others is that a better tradeoff between flexibility and expressiveness on the one hand, and control and safety, on the other hand, can be found. Tough, flexible control delegation requires investigating new approaches to composition that go beyond current technologies.

The ECF architecture targets safe dynamic and distributed deployment and refinement of service collaborations.

The paper is structured as follows. Section 2 discusses service orchestration and choreography. Section 3 lists the requirements for executable choreographies, and presents the ECF stack. Section 4 proposes a metamodel for executable choreography, and section 5 discusses the mapping to a Petri net model. Section 6 presents the CASS distributed aspect platform and section 7 discusses dynamic deployment. Section 8 deals with the enforcement of quality rules. Finally, section 9 discusses related work, and section 10 concludes this paper.

## **2 Service Orchestration and Choreography**

Process-Aware Information Systems are concerned with the composition and coordination of services and resources within or across various domains. Examples of PAIS's are Business Process Management (BPM), Business-to-Business (B2B) or Enterprise Resource Planning (ERP). The difference between BPM, B2B or ERP lies not only in the different application domains they target, but also whether the process is controlled by one control domain or more. Service composition can be seen from two main viewpoints: orchestration and choreography.

## 2.1 Orchestration and Choreography

Choreographies specify the observable sequence of messages used by businesses to exchange information with others. Participants in a choreography are peers, there is no center of control. Orchestration specifies the process being followed by an individual business or system within a domain of control. The separation between the internal aspects and the public aspect of system collaboration has two major advantages. First, businesses do not have to disclose their internal processing and data management mechanisms. Second, it decouples the observable sequence of message from its implementation within a domain of control.

The Choreography Description Language (WS-CDL) [15] has recently been adopted as the web service standard for choreography description. A WS-CDL specification describes the observable behavior of collaborations between services. It defines a global contract between participants in terms of ordering conditions on message exchanges. WS-CDL is not a programming language; it is not executable.

The orchestration viewpoint describes the behavior that a service provider performs internally within a collaboration. The Business Process Execution Language (WS-BPEL) [16] is a programming language for implementing the execution logic of a business process based on interactions between the process and its partners. A WS-BPEL process defines how multiple service interactions with partners are coordinated internally to achieve a business goal. Only one type of choreography can be executed with a single BPEL engine: master-slave choreographies. The BPEL engine controls all message exchanges between partners.

WS-CDL is used at design time to define a mutual contract between services that are under the supervision of different domain controllers. Each party can then use the global definition to build internal solutions that conform to it. Participants can also use the choreography specification to verify internal processes are conforming to a global choreography. At runtime, it can be used to detect exceptions.

The implementation of the internal process is decoupled from the global contract, and can evolve independently of the other control domains. Nevertheless, the process of modifying a choreography is very painful. Business analysts of the different organizational domains have to agree on a new global contract before local executable processes can be deployed.

## 2.2 Limitations of CDL+BPEL composition

Decentralized compositions implemented using the CDL+BPEL model are very inflexible. The global contract between the different service providers is expressed at a very low level. The contract rules do not define what is allowed within the collaboration or not in terms of collaboration semantics. They are rather expressed in terms of the observable sequence of messages. Once the global view has been agreed on, it is very hard to change it, even if the modification does not violate any essential property of the involved parties.

These restrictions prohibit the establishment of on-demand cross-domain collaboration. Ideally, the contract between participants could be specified in a form that is

independent of particular applications. They could specify what kinds of collaborations are allowed between domains independently of the specific message sequences, so that cross-domain applications can evolve without requiring new agreements to be negotiated between parties.

Some service collaborations only make sense in a particular context, and have a short life cycle. It is therefore unpractical to require the involved domain controllers to reach an agreement for the sole purpose of a choreography that has a limited number of users, or a very short life cycle. Hence, there is a need for mechanisms that support dynamic deployment and refinement of distributed collaboration.

On-demand distributed service composition requires some sort of executable choreography language. An executable choreography is not a contract that needs to be agreed on anymore, but it is an executable service composition that is controlled by the partner that initiates the collaboration application. Making choreographies executable requires the capability to delegate control over a business process to the composition partners of the process. In real world scenarios, domain controllers do not want to execute foreign composition logic. As a consequence, service composition are more sequential than they could be, BPEL-based master-slave composition is still the state-of-the-art service composition approach, and RPC is still the most common service interaction mechanism.

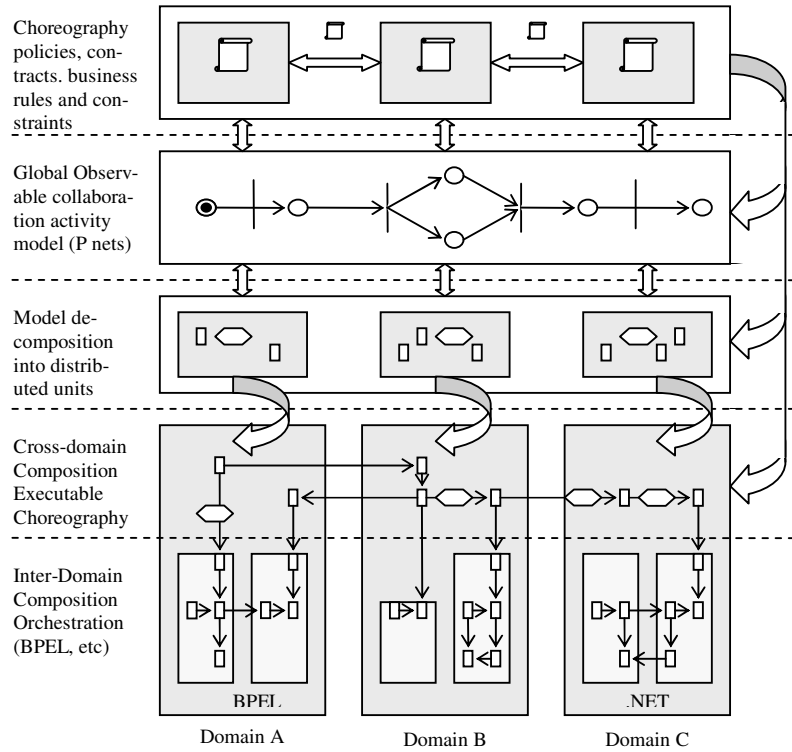
The position of the authors' is that a flexible solution to distributed composition is mandatory for true on-demand service composition. Service-Oriented Architecture (SOA) is not only about providing resources on demand, but also about resource connectivity and the establishment of service collaborations on-demand. It should be possible to dynamically distribute any service composition that can be defined using the master-slave BPEL approach in a safe way.

Distributed composition requires the contracts between the participant domains to be defined at a higher level of abstraction that leaves more space to the dynamic establishment and refinement of collaborations. The contracts would not be encoded in terms of particular message exchanges, but would tell more about the semantics of the relationships between domains. Business rules would specify what kind of relationships between domains are allowed or are not allowed.

### **3 Executable Choreography Framework**

The Executable choreography framework needs to address the following requirements. First, the control and data flow has to remain under control of the initiator of the service collaboration activity. Domain controllers have to allow the execution of composition logic on their domain that they do not control completely. Second, safety should not be compromised. Domain controllers should be able to verify the foreign logic does not violate essential properties of the system they control. Third, the entity that controls the collaboration can only affect the control and data flow of a collaboration it initiated itself. The collaboration controller can kill itself, but cannot affect other activities. Fourth, the correctness of the global composition should be verifiable; composition components scattered over several domains should be traceable back to a

global composition model. Finally, non-functional concerns and context-sensitive behaviors need to be factored out of the core logic of collaborations. Separation of concerns facilitates the runtime evolution and refinement of collaboration activities.



**Fig. 1.** Executable choreography framework stack

Fig. 1 shows the different ECF modeling and implementation layers. The lower layer represents composite services that are composed using centralized workflow mechanisms, such as BPEL engines. Executable choreographies are realized by deploying composition and transformation units across different domains of control. ECF uses an Aspect-Oriented Programming (AOP) [1,2] language to weave composition logic at the level of the message processing engine. These units work together to carry out the global model of the choreography process. These composition elements should be traceable back to a global business process definition, which can be formally checked for correctness. A formal relationship between the global model and the executable composition elements needs therefore to be defined. Tough, formal treatment of the model is out of the scope of this paper.

The upper layer represents the global contracts and business rules across and within domains of control. These contracts establish constraints on the choreography model and implementation. When deploying or refining a choreography, new relationships

between control domains can be created. Administrative domains should be able to enforce constraints on those relationships, so that the choreography does not violate the mutual business rules of participants. Contracts between controllers take the form of a set of reactive rules. The contracts do not specify the specific ordering of messages; they are application independent. Those business rules imperatively need to be enforced at the level of the executable choreography process, to ensure safety of the participants. The treatment of inter-domain and cross-domain business rules is out of the scope of this paper. Though, the paper discusses platform support for the enforcement of constraints on the composition.

Next sections establish the relationship process analysis models, executable choreography elements and a choreography process meta-model.

#### 4 Choreography Process Metamodel

Fig. 2 shows a UML representation of the choreography process metamodel. An executable choreography process is composed of collaboration activities. Collaboration activities encapsulate a series of service activities and interactions between those activities that take place in a common collaboration activity context (CAC). The CAC uniquely defines the scope of collaboration activities.

A collaboration activity realizes a logical goal, a self-contained concern within a choreography process. The focus of ECF is the composition of collaboration activities. Collaboration activities are composed by superimposition [3]. Superimposition is appropriate for adding functionality to a system in stages. Superimposition of collaborations is comparable to Collaboration-Based Design [4,5] approaches. The locus of superimposition is a set of interaction points between activities, where messages are transformed and coordinated. An interaction point is a point where messages are received or sent.

An activity is an execution of distributed computation logic that similar may cut across different control domains. In order to distribute the composition logic across different domains, the mechanism that implement the choreography control and data flow needs to be decomposed into atomic composition elements that can be localized within a domain. As opposed the CDL or BPEL ordering structures, the ECF needs to differentiate between the point at which activities are enabled, and the point at which they are terminated, because an activity might be spawned on one domain, while terminated on another one. An activity has exactly one entry point, where it is spawned and one exit point, where it terminates. An activity is spawned by sending a request message to its entry point, and terminates by emitting a response message.

An interaction is a message exchange between activities. An interaction consumes messages at activity exit points and feeds messages to activity entry points. An interaction may therefore span multiple control domains.

An interaction contribution point models localized elements of an interaction. As opposed to the other elements of the metamodel, interaction contributions are associated to a single domain of control. The CAC can be edited at runtime by interaction contributors, within a collaboration activity.

Interaction contributions are the locus of collaboration activity superimposition ECF supports two kind components that provide interaction contributions: composition units and transformation units. While composition units act upon the control flow of a collaboration activity, transformation units define its dataflow.

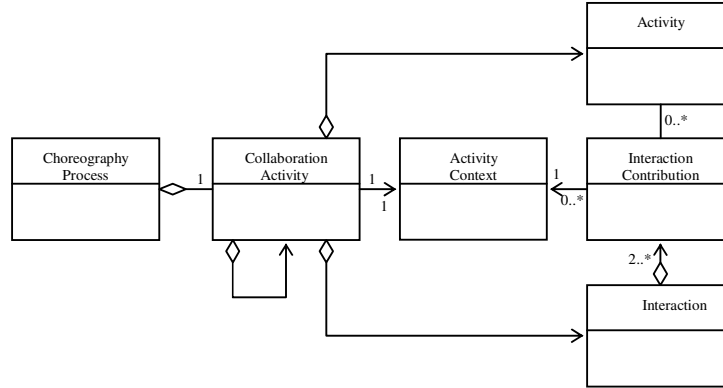


Fig. 2. Executable choreography process metamodel

## 5 Choreography Process Analysis

There is an ongoing debate in the web service composition community on whether workflows are better modeled using Petri nets or Pi-Calculus [12]. BPEL and CDL claim to be based on Pi-Calculus. Nevertheless, no analysis or validation tool supports those claims. Much work has also been done on describing web service composition using Petri nets [7]. The main reason for adopting Petri nets for web service composition modeling is the abundance of analysis techniques and tool support.

The analysis model builds on previous work in modeling web service composition with Petri nets [7]. Web Service behavior can be seen as a partially ordered set of operations, where operations are modeled as transitions and the state of the service is modeled by places. Arrows between places and transitions specify causal relations. The existence of an arc between service S and transition T is expressing using the  $W(S,T)=1$  relationship. We extend the model as to provide explicit support for control domains. The id of a control domain of a service is denoted using the *dom* function. A transition always has one controller domain and one controllee domain, denoted by the *cler* and *clee* functions.

$$cler(T) = P \Leftrightarrow \begin{cases} \exists S : W(S, T) = 1 \\ dom(S) = P \end{cases} \quad clee(T) = Q \Leftrightarrow \begin{cases} \exists S : W(T, S) = 1 \\ dom(S) = Q \end{cases}$$

The behavior of a service is modeled using a Petri net with one input place and one output place. The input place absorbs information, while the output place emits information. An activity is modeled by a net with exactly one input place and one output place. A net that represents the behavior of a service is an activity. The definition of an activity is intentionally very loose. This makes it possible to map the Petri net of

the global view of a composition to different activity diagrams, hence, allowing a service composition to be seen from different perspectives.

Composite services map to composite activities. The behavior of a non-composite service is an atomic activity. It cannot be further decomposed into smaller activities. Atomic activities take place within a single domain of control.

An interaction is a subnet which connects activities. An interaction may have several input places and several output places. An interaction's input places map to activity output places, and its output places map to activity input places.

An interaction may span over several domains of control. An interaction is composed of interaction contributions, which execute on a single domain of control. Interaction contributions may have several input and output places. A choreography process is modeled as set of superimposed collaboration activities that form a single service net. A collaboration activity is a set of service nets. The choreography process interconnects the service nets of its collaboration activities by refining their interactions.

Composition can be expressed both in terms of service entity composition and activity composition. Composition in terms of service entities is treated in [7].

We adopt the following notations:  $\varepsilon$ ,  $\circ$  represents an empty activity. It is an activity performed by the empty service  $\varepsilon$ . The empty service performs no operations.  $I.i$  represents the identity activity. It is an activity performed by the identity service  $I$ . The identity service performs a null operation  $i$ . The emitted information is the same as the absorbed information.

For service composition modeling, we adopt the standard service composition operators, such as sequential or parallel operators. Also, we adopt the service refinement operator [7].  $ref(S_1, a, S_2)$  is a service composition operator that expands the transition labeled  $a$  within service  $S_1$  by a refining service  $S_2$ . The operation modeled by the transition where the refinement is applied is replaced by a more detailed non-empty service. Refinement is the transformation of a design from a high level form to a lower level, more concrete form. It allows hierarchical modeling. We introduce the transposed operation denoted by  $fer(S_i, i)$  where service  $S_i$  is replaced by the identity service  $I$ , whose identity operation is labeled  $i$ .

For activity composition modeling, we adopt the following notation. We use the letters  $A, B, \dots$  to denote activities, and the letters  $S, T, \dots$  to denote services. The statement  $A_a^b = (S.m)_a^b$  defines an activity  $A$  which represents the execution of operation  $m$  on service  $S$ . The activity is initiated on domain  $a$  (an entity within domain  $a$  send a request message on operation  $m$  of service  $S$ ) and terminates on domain  $b$  (a response message is sent to an entity of domain  $b$ ).

An activity has exactly one entry transition, we denote by  $T_{S,m}^i$ . Similarly, an activity has exactly one exit transition,  $T_{S,m}^o$ . Interactions are refined by composing activity transitions. Activity transitions are composed into interaction contribution by introducing an intermediary place between the transition, represented by the  $*$  operator. The transition composition operator is labeled with the id of the control domain that implements the composition. Interaction contributions are therefore well localized within a domain. For example, a sequential composition of activities that span multiple domains can be decomposed into 4 interaction contribution, where  $o_\varepsilon^o$  represent

the output transition of the empty service and  $i_I^i$  represents the input transition of the identity service.

$$B_a^b \bullet C_c^d \Leftrightarrow \begin{cases} (o_\varepsilon^o * T_B^i)_a \\ (T_B^o * i_I^i)_b \\ (i_I^o * T_C^i)_c \\ (T_C^o * o_\varepsilon^i)_d \end{cases}$$

Figure 3a and 3b illustrate a service composition that spans 4 domains of control and show how a BPEL composition differs from an ECF service composition.

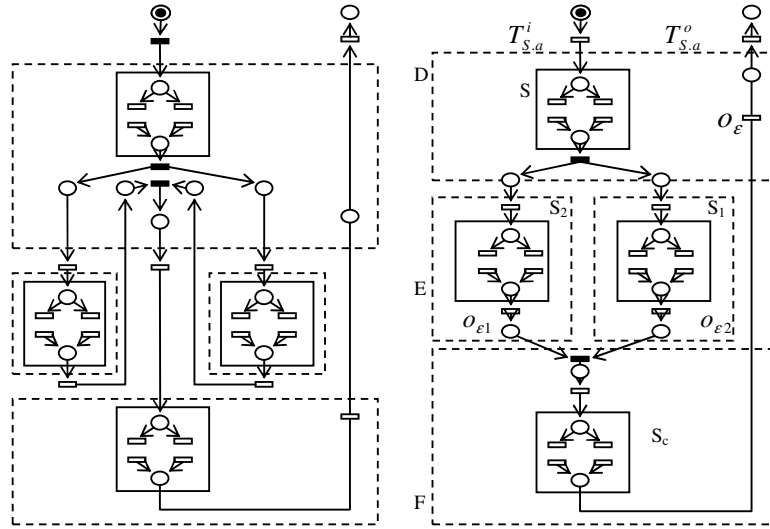


Fig. 3. Petri net-based models for BPEL master/slave choreography and ECP choreography

## 6 Contextual Aspect-Sensitive Services

In SOAP-based service architectures, non functional concerns, coordination and activity lifecycle are implemented by deploying message handlers within a handler chain. Message handlers intercept messages between the point they enter the container and the point at which the service provider is invoked.

The handler chain is therefore the right place to enforce additional constraints on service invocations, and to modify the control and data flow of service compositions. In order to support dynamic refinement, dynamic message interception needs to be supported. ECF uses an Aspect-Oriented Programming (AOP) language to weave composition logic at the level of the message processing engine. Service requests and responses are filtered and matched against a regular expression that specifies which messages should be intercepted, under which conditions. ECF uses the Contextual Aspect-Sensitive Service (CASS) [9,10,11] distributed pointcut language to specify the locus of activity superimposition.

CASS is a distributed aspect platform for service-oriented environments. A CASS pointcut expression defines a set of message interceptors that are responsible for intercepting the messages that match a message pattern expression. SOAP messages are intercepted at the level of the SOAP message processing engine, both at the client and the service side. CASS includes a rich set of mechanisms to transform, compose, synchronize and multicast the intercepted messages to remote service methods, which play the role of aspect advices. The platform also includes a native context propagation mechanism. 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. ECF associates a collaboration activity to each of those contexts. Finally, CASS supports the runtime extension of entity interfaces. Role services are services that aim at being deployed in association with other services in order to support a new interaction pattern. The role extension mechanism essentially solves an entity identity problem. A role that is associated to an entity can be referenced using the handle of the service it is attached to. These constructs allow CASS to implement collaboration activity superimposition. An interaction can be augmented at runtime by a new interaction contribution, by binding a service method to an intercepted message before, around or after it is allowed to proceed.

This section presents the CASS language constructs, and establishes their relationship with the Petri net model introduced in the previous section.

## 6.1 Control flow Refinement Constructs

Messages are intercepted at the level of the message processing engine. The current CASS implementation acts upon the Axis [17] engine. 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. Global axis 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.

CASS uses a standard aspect-oriented programming language to intercept the message context at different points of the handler chain. The current CASS implementation uses the AspectWerkz [19] framework, but any other AOP language or framework could have been used. 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.

CASS pointcut expressions represent a set of activities. The CASS pointcut interpreter translates a declarative pointcut expression that is defined with respect of WSDL definitions into local aspects that acts upon the message handlers.

The code sample of Fig. 5 defines a pointcut on calls to operations of the “Math-Service” on IIT application servers. It represents all the activities that start when an operation is called on the MathService and terminate when a response is returned. The

use of variables in CASS specification enables the runtime evolution of aspects, as well as topology independent definitions. It also avoids having to hardcode service references into choreography specifications.

```
<pointcut name="MathPointcut" type="client" service="MathService"
  operation="*" host="http://www.iit.edu:8080 OR $iithost"/>
```

The action to be performed on an intercepted message is specified in advice definitions that are bound to the pointcut expressions. Advice definitions specify the host, service and operation to which the intercepted message must be dispatched. Advice definitions can be composed and bound to multiple pointcut expressions. 3 different types of actions can be performed. An activity creation advice, denoted by  $\rightarrow A$  starts a new activity  $A$  but does not wait until the activity completes. An activity termination advice, denoted by  $\leftarrow A$ , waits until activity  $A$  terminates. An activity creation/termination advice, denoted by  $\leftrightarrow A$ , starts a new activity  $A$  and waits until it terminates. Activities can be refined at 3 different points in their lifecycle:

- a. *Before refinement.* In terms of activities, the following before refinement expression is interpreted as “before activity  $S.a$ , activate activity  $A$ , and wait until it terminates”.

$$before_D(S.a) : \leftrightarrow A \Leftrightarrow \begin{cases} (T_A^i * T_{S.a}^i)_D \\ (T_{S.a}^i * T_A^i)_D \end{cases} \quad D \in cler(T_{S.a}^i) \cup clee(T_{S.a}^i)$$

In terms of service composition, it can be interpreted as a service refinement  $ref_D(S, T_S^i, S_a)$ , where the input transition  $T_S^i$  is expanded into service  $S_a$  when the  $a$  operation is invoked in service  $S$ . The *before* keyword is annotated with the id of the control domain that executes the refinement. If the refinement is applied to a client-side pointcut expression, the control domain is the controller of the input transition of the service (similar to a ‘call’ pointcut in aspect languages). If the refinement is applied to a service side pointcut expression, the control domain is the controller of the input transition of the service (similar to an ‘execution’ pointcut in aspect languages).

- b. *After refinement.* In terms of activities, the following after refinement expression is interpreted as “after activity  $S.a$ , wait until activity  $A$  terminates”.

$$after_D(S.a) : \leftarrow A \Leftrightarrow \begin{cases} o_e * T_A^i \\ T_{S.a}^o * (T_{S.a}^o \parallel T_A^o) \end{cases} \quad D \in cler(T_{S.a}^o) \cup clee(T_{S.a}^o)$$

- c. *Around refinement.* In terms of activities, the following around refinement expression is interpreted as “instead of activity  $S.a$ , activate activity  $A$  and activate activity  $S.a$  when a special transition called ‘*proceed*’ occurs within activity  $A$ ”.

$$around_D(S.a) : \leftrightarrow A \Leftrightarrow \begin{cases} T_{S.a}^i * T_A^i \\ proceed_A^o * T_{S.a}^i \\ T_{S.a}^o * proceed_A^i \\ T_A^o * T_{S.a}^o \end{cases} \quad D \in dom(S) \cup cler(T_{S.a}^i)$$

In terms of service composition, it can be interpreted as a service refinement  $ref_D(fer(S,i), i, ref(S_a, proceed, S))$ . The refined service is collapsed into the identity service  $I$ , which is in turn refined into service  $ref(S_a, proceed, S)$ , which represents a refinement of service  $A$ , whose 'proceed' operation is expanded into the original service  $S$ . The 'proceed' operation has the same signature as the refined operation. If the 'proceed' operation is not executed within the refinement activity, the initial activity is not executed at all.

The code sample of Fig 6 illustrates a simple CASS 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 refinements.

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

In CASS, composition units are defined by composing pointcuts and advices. Composite activity creation structures are defined by composing advices. The *par* advice composition keyword spawns or terminates one or more activities executing concurrently.

$$after_E(S.a) : \rightarrow par_D(\rightarrow S1.b, \rightarrow S2.c) \Leftrightarrow \begin{cases} T_{S.a}^o * (T_{S.a}^o \parallel (T_{S1.b}^i \parallel T_{S2.c}^i)_D)_E \\ T_{S1.b}^o * o_{\epsilon 1} \\ T_{S2.c}^o * o_{\epsilon 2} \\ o_{\epsilon} * T_{S.a}^o \end{cases}$$

The exclusive choice activity composition structure is defined using a similar notation with the *choice* keyword:  $choice_F(\rightarrow S1.b, \rightarrow S2.c)$ . The following code samples shows how the parallel composition unit is defined in CASS.

```
<pointcut name="MathServiceAdd" .../>
<parallel name="beforeaddflow" bind-to="MathServiceAdd" type="after">
  <advice name="add80" ... />
  <advice name="add81" ... />
</parallel>
```

Composite activity termination structures are defined by composing pointcut expressions. The  $after_F(par_F(\leftarrow S1.b, \leftarrow S2.c))$  pointcut composition expression is triggered after the activity composed of activities  $S1.b$  and  $S2.c$  terminates. It is a *join* pointcut. The discriminator activity termination structure has a similar notation using the *disc* keyword. The discriminator waits for the termination of one of the composed activities before activating the activity advice. The following code samples shows how the join composition unit is defined in CASS.

```
<join name="joinBeforeMath" context="flow.MathServiceAdd">
  <pointcut name="joinMath1" ... />
  <pointcut name="joinMath2" ... />
</join>
<advice name="joinAdvice" type="after" bind-to="joinBeforeMath" ... />
```

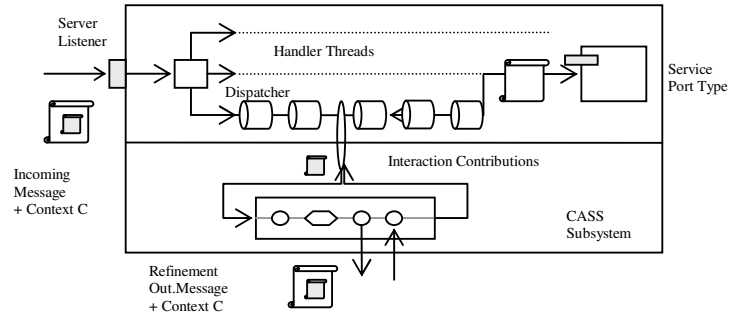
## 6.2 Data Flow Refinement Constructs

WS-CDL is not designed to enable choreographies to be specified from existing WSDL Web Service interfaces. It merely declares a choreography of operations that can be specified in some WSDL once the choreography is implemented. The ECF targets the runtime deployment of new choreographies. Hence, the CASS pointcut expressions are defined with respect of existing WSDL interfaces. In multi-party peer-to-peer interaction scenario, an arbitrary set of interfaces does not necessarily lead to a possible collaboration. CASS therefore supports a message transformation component, the transformation unit, whose role is to adapt the process messages to existing WSDL's. Transformation units can transform both the structure and the content of the intercepted messages. It uses XSLT to generate a new SOAP message that conforms to the interface of the remote 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. Transformation units can also maintain state within collaboration activities and buffer messages. Those variables can be used within the XSL templates, to transform messages according to the state of the collaboration activity, or to aggregate messages.

## 6.3 Context Propagation

Choreographies are defined and deployed by the domain of control that takes advantage of the added value resulting from the choreography (just as for BPEL master/slave choreographies). The composition should therefore only apply to interactions that are part of the collaboration activities of the choreography. Hence, there is a need for a mechanism to unambiguously discriminate interactions according to the collaboration activities they take part in. CASS passes collaboration activity contexts (CAC) in the header of messages. Context propagation allows pointcuts to be defined on a per-activity basis and enable concurrent service customization [14]. CAC's are propagated along the interactions of an activity by keeping a CAC reference within the server message processing threads. The activity context uniquely defines the scope of a collaboration activity.

When a message comes in at the server listening port, it is dispatched to a message handler thread. The thread inspects the message and executes the corresponding message handlers, before executing the service implementation. When a message is intercepted, the composition logic is executed within the thread of the message handler chain. If the composition logic results in the invocation of another service, the invocation message is assigned the same activity contexts then the incoming message. The activity context is also propagated to new threads that are spawned by composition units. Message can be assigned several contexts. Activity contexts allow message correlation. Correlated messages are part of the same collaboration activity.



**Fig. 4.** Collaboration Activity Context propagation

New activity contexts are declared at the pointcut level. A contextual pointcut defines the set of message processing points where new collaboration activities start. Each time an event triggers a contextual pointcut, the intercepted message is assigned a new CAC. The CAC carries the following information:

- a. *Collaboration activity name.* The context name is the name of the collaboration activity. It is the name of the pointcut that defines the context.
- b. *Collaboration activity id.* A different id is attributed to each context joinpoint instance. It is the instance reference of the collaboration activity.
- c. *Joinpoint callback.* The URI of the callback interface of the joinpoint that started the collaboration activity instance. The joinpoint callback is used to terminate a collaboration activity at the point at which it started. It can be used to implement transaction compensation.
- d. *Collaboration activity data.* The data associated to the activity. The context data can also be maintained by an external service. In the later case, only a URI to the service data is propagated. Composition and transformation units can access the activity data and edit it to update the state of the collaboration activity.

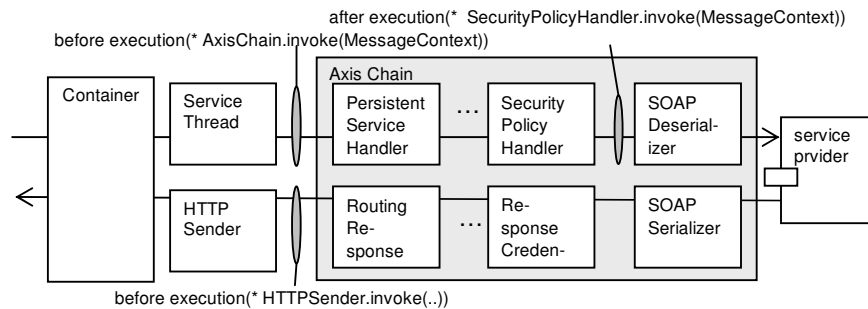
## 7 Dynamic Deployment

The CASS platform includes a dynamic deployment service. A client parses the choreography specification, and partitions it into composition and transformation units. The deployment service translates those descriptors into low-level pointcut expressions that act upon the message handlers. All pointcuts are associated to an CAC. 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 collaboration activity 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.

## 8 Enforcement of Quality Rules

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.

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. There are 3 places where superimposition constraints can be enforced. Rules can constrain which messages can be intercepted, at what point in the message processing chain, and in what context. Rules can also constrain the kind of composition actions that are allowed within the domain, in a given context. Finally, the binding between the interaction points and the composition logic can be constrained.



**Fig 5.** Message Context interceptors – CASS pointcut expressions are translated into lower level pointcut expressions that act upon the message handlers. Depending on the handler they affect, CASS pointcuts can be attributed quality properties

## 9 Related Work

Symphony [6] proposes an algorithm for partitioning BPEL processes and a platform for executing “distributed orchestrations”. The approach demonstrates significant performance improvements over centralized orchestrations in terms of increased throughput, scalability and response time. The authors acknowledge the decentralization increases the complexity of the system, especially regarding error handling. The ECF relates to the work presented in the paper in that the coordination aspects are deployed in a decentralized way. However, Symphony does not take into account cross-domain compositions, and does not support dynamic deployment and refinement of collaborations.

The Web Service Management Layer (WSML) [8] takes advantage of the dynamic AOP language JAsCo to provide a highly dynamic client-side web service management environment that decouples service management concerns from the client applications. The WSML does not intercept SOAP messages. It rather intercepts calls to

the SOAP engine, within the client application code. The WSML does not address service choreography nor advanced service composition.

The Web Service Composite Application Framework (WS-CAF) [18] is a specification that guides the development of advanced composite Web Services by specifying standard interfaces for coordination, activity life-cycle and transaction management concerns. WS-CAF acknowledges the need for a generic context-passing mechanism and explicit support for activity lifecycle management. WS-CAF is inspired by the WS-Coordination and the WS-Transaction specifications, which propose similar mechanisms. Many concepts of WS-CAF can be retrieved in CASS. Yet, in WS-CAF, activity contexts are not propagated transparently. Context propagation is driven by the participating entities. Furthermore, WS-CAF is very inflexible. WS-CAF requires applications to implement a set of interfaces that allow them to register with a coordination service and an activity life-cycle service. WS-CAF does not allow collaboration between services to be deployed or refined at runtime. The ECF allows existing services to be coordinated non-invasively, without requiring specific interface support. On the other hand, the ECF requires platform support.

Further work on the ECF includes a quantitative comparison of ECF choreographies with CDL+BPEL+CAF choreographies, integration of ECF with Petri net-based modeling tools, and linguistic support for domain collaboration contracts. Semantic Web techniques will most probably be required for the specification of global quality contracts.

## **10 Conclusions**

It is the author's position that current service composition approaches are too inflexible. They prohibit on-demand establishment of collaborations between services. The Executable Choreography Framework (ECF) acknowledges the need for collaboration contracts between control domains, but argues that those should be expressed in an application-independent way, so that collaborations can evolve without requiring intervention of business analysts from the participating domains. This work establishes the first steps towards an Executable Choreography Language, for the dynamic deployment and refinement of peer-to-peer collaborations. Requirements for on-demand collaboration are identified, and an executable choreography stack that addresses those requirements is proposed. A metamodel for choreography processes is presented. The ECF uses the CASS distributed aspect platform to enable dynamic superimposition of collaboration activities. ECF defines a relationship between the distributed CASS pointcut language constructs and Petri net-based collaboration models, so that the correctness of ECF choreographies can be checked for using existing service composition modeling tools.

## **Acknowledgements**

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

## References

1. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. Proceedings of the European Conference on Object-Oriented Programming, Springer-Verlag (1997)
2. Filman, R., Friedman, D.: Aspect-oriented Programming is Quantification and Obliviousness. Workshop on Advanced Separation of Concerns, OOPSLA 2000 (2000)
3. Katz, S: A superimposition control construct for distributed systems, ACM Transactions on Programming Languages and Systems (TOPLAS), ACM Press (1993)
4. VanHilst, M., Notkin, D.: Using Role Components to Implement Collaboration-Based Designs. Proceedings of the 11th ACM conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press (1996)
5. Smaragdakis, Y., Batory, D.: Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. ACM Transactions on Software Engineering and Methodologies, ACM Press (2002)
6. Chaffe, G., Chandra, S., Mann, V., Nanda, M. G.: Decentralized Orchestration of Composite Web Services. Proceedings of the Thirteenth International World Wide Web Conference, New York, NY, USA, ACM Press (2004)
7. Hamadi, R., Benatallah, B.: A Petri Net-Based Model for Web Service Composition. Proceedings of the 14th Australasian Database Conference (ADC'03), Australian Computer Society, Adelaide, Australia (2003)
8. Verheecke, B., Cibrán, M. A., Jonckers, V.: Aspect-Oriented Programming for Dynamic Web Service Monitoring and Selection. In Proceedings of the European Conference on Web Services, Erfurt, Germany. LNCS 3250 Springer (2004)
9. Cottenier T., Elrad T, Prunicki, A.: Contextual Aspect-Sensitive Services, formal demonstration presented at the 4th International conference on Aspect-Oriented Software Development (AOSD'05), Chicago, USA (2005)
10. Cottenier, T., Elrad, T.: Dynamic and Decentralized Service Composition with Contextual Aspect-Sensitive Services, First International Conference on Web Information Systems and Technologies, Miami, USA (2005)
11. Cottenier, T., Elrad, T.: Validation of Aspect-Oriented Adaptations to Components. Ninth International Workshop on Component-Oriented Programming as part of ECOOP'04, Oslo, Norway (2004)
12. Van der Aalst, W.M.P.: Pi calculus versus Petri nets: Let us eat "humble pie" rather than further inflate the "Pi hype" (unpublished discussion paper) (2004)
13. Van der Aalst, W. M. P.: On the automatic generation of workflow processes based on product structures. Computer in Industry, Vol. 39, No. 2, Elsevier Science (1999)
14. E. Truyen.: Dynamic and context-sensitive composition in distributed systems, Ph.D. Thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, (2004)
15. Web Services Choreography Description Language (WS-CDL) Version 1.0, W3C Working Draft 17, <http://www.w3.org/TR/ws-cdl-10/> (2004)
16. Business Process Execution Language for Web Services, BEA Systems, IBM, Microsoft, SAP AG and Siebel Systems <http://www-128.ibm.com/developerworks/library/ws-bpel>
17. Apache, Axis homepage <http://ws.apache.org/axis> (2000)
18. Web Services Composite Application Framework (WS-CAF). Arjuna Technologies Ltd., Fujitsu Limited, IONA Technologies Ltd., Oracle Corporation, and Sun Microsystems, Inc, <http://developers.sun.com/techtopics/webservices/wscaf/primer.pdf> (2003)
19. Aspectwerkz homepage <http://aspectwerkz.codehaus.org/> (2004)