



# D13.9v0.1 WSMX Choreography

WSMX Working Draft 15 June 2005

**This version:**

<http://www.wsmo.org/TR/d13/d13.9/v0.1/20050615/>

**Latest version:**

<http://www.wsmo.org/TR/d13/d13.9/>

**Previous version:**

<http://www.wsmo.org/TR/d13/d13.9/v0.1/20050606/>

**Editor:**

Armin Haller

**Co-Authors:**

Thomas Haselwanter

James Scicluna

This document is also available in a non-normative [PDF](#) version.

## Table of contents

### 1 Introduction

- [1.1 Overview](#)
- [1.2 Purpose of this document](#)
- [1.3 Document Overview](#)

### 2 WSMX Choreography

- [2.1 The choreography component in the big picture](#)

- 2.2 Component Functionality

### 3 WSMX Choreography Syntax

- 3.1 Choreography Interface Syntax Basics
- 3.2 Meta Information
  - 3.2.1 Namespace References
  - 3.2.2 Header
- 3.3 Choreography Interface Specification
  - 3.3.1 Guarded Transitions
  - 3.3.2 State Signature
- 3.4 Logical Expressions in Choreography Interface Specification

### 4 Architecture

- 4.1 Core
- 4.2 Execution Process

### 5 Implementation

- 5.1 Basic Class Diagrams
- 5.2 Back-End Rule Engine

### 6 Related Work

- 6.1 Choreography and Orchestration Languages
- 6.2 ASM Tools
- 6.3 Rule-Based Systems

### 7 Future Work

### 8 Conclusion

### References

### Appendix A: Human-Readable Syntax

- [A.1. BNF-Style Grammar](#)
- [A.2. Example of Human-Readable Syntax](#)

## [Appendix B: Glossary](#)

# 1. Introduction

## 1.1 Overview

The Web Services Execution Environment (WSMX) [[Cimpian et al., 2005](#)] is an execution environment for dynamic discovery, selection, mediation and invocation of semantic Web Services. WSMX is based on the Web Services Modeling Ontology (WSMO) [[Roman et al., 2005](#)] which describes all aspects related to the discovery, mediation, selection and invocation of Web Services. WSMX is a reference implementation for WSMO. The goal is to provide both a test bed for WSMO and to demonstrate the viability of using WSMO as a means to achieve dynamic inter-operation of semantic Web Services.

## 1.2. Purpose of this document

After discovering a Web Service and its respective service description, one has to know the observable behaviour of the Web Service in order to achieve the desired functionality. WSMO [[Roman et al., 2005](#)] is the first service modelling languages clearly distinguishing the terms choreography and orchestration and providing means to express them in its service description. Other standards to define the exchange sequence only emerged on top of WSDL descriptions to solely describe business processes of composed web service. The most prominent work in this domain represents BPEL4WS [[BPEL4WS, 2003](#)], which distinguishes two ways of describing the business processes: executable business processes and abstract business processes. The first models the internal behaviour of a partner role in an interaction, while the second describes the message exchange behaviour of the involved parties. A different approach is followed by WS-CDL [[WS-CDL, 2004](#)], which only defines a non executable message exchange between partners in a Web Service collaboration. This corresponds to the abstract process definition in BPEL4WS [[BPEL4WS, 2003](#)].

Choreography and Orchestration in WSMO [[Roman et al., 2005b](#)] are part of the

interface definition of a WSMO service description [Roman et al., 2005] and describe how to communicate with the service such that the service will provide its capability and how the service collaborates with other WSMO services to achieve its capability. Within this document we will define the syntax and semantics of the WSMO choreography interface and add concepts to the conceptual model to extend the abstract choreography description in WSMO to an executable one. To show the functionality of the component, we define a scenario describing a collaboration between two parties. The scenario assumes that the existence of the service provider was known beforehand and its service description including the choreography definition is published in a repository. The scenario describes how the functionality advertised by the service provider can be consumed by the service requester by elaborating WSMX. It further imposes requirements on the functionality of the choreography component itself.

### 1.3. Document Overview

TBC

## 2. WSMX Choreography

Choreography in WSMO describes a concept aligned to the generic definition of the W3C glossary [W3C Glossary, 2004], but something different to the notion of Choreography in WS-CDL [WS-CDL, 2004] or in [Dijkman & Dumas, 2004]. In WSMO Choreography describes the behaviour of a service from one role instance. In [Dijkman & Dumas, 2004] this corresponds to the term interface behaviour. Since it is related exclusively to one role instance, the choreography in WSMO only describes send and receive events, denoted by a choreography ontology extension, called mode which can take the value `out` for send and `in` for receive events. Hence the WSMO choreography does not describe interactions between different roles. It concerns all possible interactions providing its capability with their users. Any user of a Web service, automated or otherwise, is a client of that service. These users may, in turn, be other Web Services, applications or human beings.

In the context of the choreography component in WSMX we have to further distinguish between provider and requester choreographies similar to the distinction of provided and required interface behaviour in [Dijkman & Dumas, 2004]. The distinction is based on the initial communication task the user is required to use. A provider choreography can only use communication task types where a

receive event occurs first. A reply task is not necessarily mandatory, but desirable in most cases. A requester choreography can only use communication task types where an event always occurs first, after which optional receive events may occur. The idea is that a requester choreography works as the counterpart of the provider choreography and that therefore the send event corresponds to the request event and vice versa. Only if the requester and provider choreography are perfectly symmetric a direct interaction is possible. It has also to be noted that the provider choreography defines the complete behaviour of a particular service. The requester choreography on the other hand might describe the complete behaviour or it might only describe the required behaviour for one specific interaction if the service provider is already known.

## 2.1 The choreography component in the big picture

The current WSMX system supports four entry points, whereas currently one, namely the "Web Service execution with choreography", requires to call the choreography component in its execution semantics. The following entry-point initiates this execution semantic:

```
receiveMessage(OntologyInstance,WebServiceID, ChoreographyID):  
ChoreographyID
```

Once the service requester knows which Web Service to invoke, this entry point provides the means for the back-and-forth conversation between the service requester, WSMX and the service provider in order to achieve the functionality expected by the service requester. The service requester is providing all the necessary data to invoke the service by giving fragments of ontology instances (e.g. business documents such as catalogue items or purchase orders in its own ontology). To identify the functionality, the choreography component requires to provide within the WSMX architecture, we define a collaboration scenario between one service requester and one service provider.

As mentioned above the prerequisite is that the service provider is already identified as one suitable to fulfill the request of the service requester. The means how to achieve the prior knowledge about the service provider are manifold, but of no relevance for this deliverable. We refer to the WSMO [[Keller et al., 2004](#)] and WSMX Discovery [[Lara et al., 2005](#)] for detailed descriptions on how to discover service definitions.

The service provider offers a specific functionality and the necessary semantic

descriptions as a Service. The business processes of the service provider (defining the tasks to perform for offering the functionality) are internally defined in its back end application and are out of the scope of this deliverable. It only provides the endpoints to consume the functionality of the Service and describes the required behaviour to achieve it in the interface description of a WSMO Web Service definition.

In this scenario, WSMX acts as an mediating entity between the service requester and the service provider. We assume that the service requester uses WSMX to actually invoke the desired service. It is obvious that in such case the interface definition of the service offered by the service provider has to be stored in any repository known to the WSMX applied by the service requester. This interface definition is described in the provider's choreography interface. Iff the service requester's and service provider's interface behaviour match, i.e. if the order constraints for the sequencing of the messages sent and received and the information encoded in the messages by the requester match the sequence of messages and its information received and sent by the provider, these two can collaborate. Their combined behaviour is a choreography in the definition of [\[WS-CDL, 2004\]](#) or [\[Dijkman & Dumas, 2004\]](#). In any other case the heterogenities between the two interface behaviour's have to be resolved by the Process Mediation component. A detailed description of the scope of the Choreography component is given in section 2.2, for more details on the functionality of the Process Mediator itself we refer the reader to [\[Cimpian & Mocan, 2005\]](#).

## 2.2 Component Functionality

Before any conversation can take place WSMX has to offer the means for the service provider to store its choreography interface. Since the choreography interface is part of the Web Service description this functionality is provided by the methods offered by the Resource Manager interface [\[Zaremba et al., 2005\]](#).

When starting a collaboration the requester will either provide its choreography definition encoded in the message or it will pass a reference to the location of the choreography definition. The choreography component provides the following method to be called at the start of any collaboration to define the roles of the partners in the collaboration and uniquely identify the collaboration itself. Choreography in WSMO deals with binary collaborations only. To define n-ary collaborations one has to define it in the Orchestration interface of the service. However in the context of the discovery and the negotiation it might be

necessary to initiate several provider choreography instances for one requesting choreography instance. To keep this extensibility the components interface allows to initiate the respective instances independently from each other via two separate methods:

for requester choreographies:

```
public void initiateChoreography(Goal goal)
```

and respectively for provider choreographies:

```
public void initiateChoreography(WebService webService)
```

Any component calling one of these two methods expects the choreography component to parse the respective Choreography Interface descriptions and to build the class model. The logical interconnection between a requesting choreography instance and its respective one to many providing Choreography Interfaces is internally managed by use of the `ConversationID` assigned to every collaboration by the system. This allows the choreography component to keep track of the state of communication on either sides, the requester and provider choreography.

```
updateState(Origin, Message):AbstractGrounding
```

For every message received by WSMX the `updateState` method of the Choreography component is called to determine the subsequent state of the opposite choreography. The `Origin` parameter defines the participant where the `Message` was received from. Hence whenever such an event occurs (i.e. a message is received) the choreography component identifies the collaboration the message is part of and its respective state. Further it determines what message exchange should conclude in a given state, i.e. it evaluates the condition in the guarded transitions of the source choreography and target choreography and checks if the updates of the opposite state transition are sufficient to conclude the collaboration.

If there is no match in the update part of the guarded transition or even earlier if there are no conditions evaluating true in the opposite choreography description the process mediation component is called to resolve heterogeneities in the two interface descriptions.

It has to be noted that the choreography component only deals with the choreographies of the involved participants but does not know anything about the concrete endpoints of the communication. It differs between the participants in the choreography who act out a certain role that is defined in this choreography. The actual grounding is dealt within the Invoker component. Hence the return value of the `updateState` method is an abstract references to a grounding information which is resolved in the component performing the lowering from the WSMML representation to the target protocol. The rationale behind this is to allow a runtime binding of choreography participants to choreography roles and to be able to reuse choreography interface descriptions.

### 3 Choreography Syntax

In this chapter we introduce the syntax of the choreography interface of a WSMO Web Service description. The syntax defined in this document represents a preliminary version applied in the implementation of the WSMX choreography component. It might be subject to changes if the syntax is defined in WSMO. However the choreography interface description represents an independent document referenced from a Web Service description. This separation ensures the reusability of choreography interface descriptions and allows to reference  $n$  number of such kind of descriptions from within one WSMML document. The syntax of a choreography interface document is in accordance to a WSMO service description by only extending it by representation means for the dynamic behaviour of the information interchange that takes place when a service is used.

A Choreography interface description document has the following structure:

```
chor      = namespace?
           definition*
definition = choreography
```

This chapter is structured as follows. According to the definition in [Bruijn et al., 2005] the Choreography description syntax basics, such as the use of namespaces, identifiers, etc., are depicted in Section 3.1 whereas with most parts it is referred to their definition in [Bruijn et al., 2005]. The elements describing the meta information of the document are described in Section 3.2 and again mostly referred to their definition in [Bruijn et al., 2005]. The

Choreography specific part is described in detail in section [3.3](#). Finally, the preliminary version of the logical expression syntax allowed in defining the interface description is specified in Section [3.4](#).

## 3.1 Choreography Interface Syntax Basics

The first part of a Choreography Interface descriptions provides similar to every other WSMML document meta-information about the specification. It contains such things as the namespace references, non-functional properties (annotations), import of ontologies and references to mediators. As with every WSMML document the ordering of this meta-information block is strict. The second part of the specification, consisting of the rules for the guarded transitions is not ordered.

For a detailed description about the use of namespaces, identifiers and datatypes in the Choreography Interface description we refer to [\[Bruijn et al., 2005\]](#).

## 3.2 Meta Information

This section describes the elements on top of a Choreography Interface description, in particular the namespace references and the header information.

### 3.2.1 Namespace References

As with every WSMML document at the top of a Choreography Interface description there is an optional block of namespace references, which is preceded by the `namespace` keyword. The `namespace` keyword is followed by a number of namespace references. For a description of the elements given below and an example we refer to [\[Bruijn et al., 2005\]](#).

```
namespace      = 'namespace' prefixdefinitionlist
prefixdefinitionlist = full_iri
                    | '{' prefixdefinition ( ',' prefixdefinition )* '}'
prefixdefinition    = name full_iri
                    | full_iri
```

---

## 3.2.2 Header

As any other WSMML specification the Choreography Interface description may contain non-functional properties, may import ontologies and may use mediators:

```
header =   nfp
          | importsontology
          | usesmediator
```

### ***Non-Functional Properties***

Non-functional properties may be used for the Choreography Interface description document as a whole but also for each element in the specification. Once more we refer to [Bruijn et al., 2005] for a detailed description of this language block and some examples.

```
nfp =   'nfp' attributevalue* 'endnfp'
       | 'nonFunctionalProperties' attributevalue* 'endNonFunctionalProperties'
```

### ***Importing Ontologies***

Ontologies may be imported in a Choreography Interface description through the import ontologies block, identified by the keyword `importsontology`. In fact any Choreography Interface has to import ontologies to be used in the rules. This import avoids the replication of the static information represented in ontologies in the Choreography Interface description. For an explanation how recursive imports are handled and for some examples we refer to [Bruijn et al., 2005] again.

### ***Using Mediators***

This optional block identified by the keyword `usesmediator` represents a reference to mediators to resolve any sort of heterogenities. In the case of the Choreography Interface description this can be `ooMediators` and `wwMediators`. The `ooMediators` have the role of resolving possible representation mismatches between ontologies, both on the conceptual and on

the instance level. The **wwMediators** connect Web Services, resolving any data, process and protocol heterogeneity between the two. More details on the use of different mediators can again be found in [\[Bruijn et al., 2005\]](#).

```
usesmediator = 'usesMediator' idlist
```

### 3.3 Choreography Interface Specification

The actual Choreography Interface specification part is identified by the **choreography** keyword optionally followed by an IRI which serves as the identifier of the choreography. If no identifier is specified for the choreography, the locator of the choreography serves as an identifier. This is followed by one or more State Signatures and one finally by one or more Guarded Transitions.

```
choreography = 'choreography' id? header* state_signature*  
guarded_transition*
```

Example:

```
choreography airJourney
```

In this section we explain the Choreography Interface modeling elements.

#### 3.3.1 State Signature

```
state_signature = 'stateSignature' id? role+
```

A State Signature definition starts with the **stateSignature** keyword, which is optionally followed by the identifier of the State Signature. Finally this is followed by one or more roles defining the mode of the state.

A Choreography Interface specification describes the dynamic behaviour of a Service and represents an extension of the Webservice specification. Hence the required ontologies are already in place when defining the Choreography Interface as they have been used to describe the capability of a Webservice already. For the dynamic behaviour of the Webservice one has to subsequently

model the states and guarded transition rules, whereas the states in the Choreography Interface are defined by all legal WSMO identifiers, concepts, relations, functions, and axioms. A state represents a stable status within the dynamics of a choreography interface specification that is existent as long as attribute values of instances are not changed. The ontologies used to represent states are imported through the `importsOntology` keyword in the header of the Choreography Specification (see section [3.2.2](#)). The elements that can change and that are used to express different states of a choreography, are the instances (and their attribute values) of concepts, functions, and relations that are not defined as being static. Each of these instances are then bound to the mode denoting the operational behaviour of the respective state (see below).

## Role

```
role = grounded_role
      | un_grounded_role
grounded_role = mode id 'withgrounding' idlist
un_grounded_role = mode id
mode = 'in'
      | 'out'
      | 'static'
      | 'controlled'
```

The role block is used to define the communicative activities to be performed on the instance data representing the state of the communication. It can be distinguished between grounded and ungrounded roles. Within the choreography only an abstract grounding is referenced. The actual physical grounding is resolved in the grounding specification of a Webservice [[Kopecky et al., 2005](#)]. Whether a role is grounded or not depends on the mode it performs. The mode can take the following values [[Roman et al., 2005b](#)]:

- static - meaning that the extension of the concept, relation, or function cannot be changed. If not explicitly defined for an element imported in the `importsOntology` statement, the attribute mode takes this value by default. Static modes can only be ungrounded since whether the service nor the environment can change any of the ontology elements.

- controlled - meaning that the extension of the concept, relation, or function can only be changed by the service. The difference to the out mode is that it represents an ungrounded role.
- in - meaning that the extension of the concept, relation, or function can only be changed by the environment. It represents a grounded role which has to reference a mechanism that implements write access for the environment.
- out - meaning that the extension of the concept, relation, or function can only be changed by the service. It also represents a grounded role which has to reference a mechanism that implements read access for the environment.

### 3.3.2 Guarded Transitions

A Guarded Transition definition starts with the `guardedTransition` keyword, which is optionally followed by the identifier of the guardedTransition. Finally this is followed by one or more rule definitions.

```

guardedTransition = 'guardedTransition' id? rule+
rule                = 'if' condition 'then' rule+ 'endif'
                       | 'choose' variablelist condition 'then' rule+ 'endchoose'
                       | 'doforall' variablelist condition 'then' rule+ 'endforall'
                       | updaterule

```

#### **Rule**

```

updaterule = modifier '{' fact
              '}'
modifier   = 'add'
              | 'delete'
              | 'update'
fact       = term attr fact? 'memberOf' termlist fact update?
              | term 'memberOf' termlist fact update? attr fact
              | term attr fact

```

```
attr_fact      =  {' attr_fact_list
                  }
attr_fact_list =  term 'hasValue' termlist fact_update?
                  | attr_fact_list ',' term 'hasValue' termlist
                  | fact_update?
fact_update    =  '=>' termlist
```

## 3.4 Logical Expressions in Choreography Interface Specification

# 4 Architecture

intro to the architecture

## 4.1 Core

We take CoreASM as the starting point illustrating what we should add/modify

## 4.2 Execution Process

Execution behaviour of the internal engine

# 5 Implementation

overview of what we used/modified/added

## 5.1 Basic Class Diagrams

overview of classes and their interfaces

## 5.2 Back-End Rule Engine

this should describe what we looked at and a rationale of our choice

# 6 Related Work

Diverse related work due to different interpretation of choreography concept.

## 6.1 Choreography and Orchestration Languages

WS-CDL, SWSL, OWL-S (?), BPEL4WS, YAWL etc.

## 6.2 ASM Tools

overview of what ASM tools are there

## 6.3 Rule-Based Systems

what's out there...

# 7 Future Work

here comes some evaluation

# 8 Conclusion

# References

**[BPEL4WS, 2003]** T. Andrews et al.: Business Process Execution Language for Web Services Version 1.1, 2003. Available from <http://www-128.ibm.com/developerworks/library/ws-bpel/>.

**[Bruijn et al., 2005]** J. de Bruijn, H. Lausen, R. Krummenacher, A. Polleres, L. Predoiu, M. Kifer, D. Fensel: The Web Service Modeling Language WSML.

*WSML Final Draft v0.2*, 2005. Available from <http://www.wsmo.org/TR/d16/d16.1/v0.2/>.

**[Cimpian et al., 2005]** E. Cimpian, T. Vitvar, M. Zaremba (editors): Overview and Scope of WSMX. *WSMX Working Draft v0.2*, 2005. Available from <http://www.wsmo.org/TR/d13/d13.0/v0.2/>.

**[Cimpian & Mocan, 2005]** E. Cimpian, A. Mocan: Process Mediation in WSMX. *WSMX Working Draft v0.1*, 2005. Available from <http://www.wsmo.org/TR/d13/d13.7/v0.1/>.

**[Dijkman & Dumas, 2004]** R. Dijkman, and M. Dumas: Service-Oriented Design: A Multi-Viewpoint Approach. *International Journal of Cooperative Information Systems* 13(4): 337-368, 2004.

**[Keller et al., 2004]** U. Keller, R. Lara, and A. Polleres (editors): WSMO Web Service Discovery, *WSML Working Draft v0.1*. Available from <http://www.wsmo.org/TR/d5/d5.1/v0.1/>.

**[Kopecky et al., 2005]** J. Kopecky, and D. Roman (authors): WSMO Grounding, *WSMO Working Draft v0.1*. Available from <http://wsmo.org/TR/d24/d24.2/v0.1/>.

**[Lara et al., 2005]** R. Lara, H. Lausen, and I. Toma (editors): WSMX Discovery, *WSMX Working Draft v0.2*. Available from <http://www.wsmo.org/TR/d10/v0.2/>.

**[Roman et al., 2005]** D. Roman, H. Lausen, and U. Keller (editors): Web Service Modeling Ontology Standard. *WSMO Working Draft v1.0*, 2005. Available from <http://www.wsmo.org/2004/d2/v1.0/>.

**[Roman et al., 2005b]** D. Roman, J. Scicluna, and C. Feier (editors): Ontology-based Choreography and Orchestration of WSMO Services. *WSMO Working Draft v0.2*, 2005. Available from <http://www.wsmo.org/TR/d14/v0.2/>.

**[W3C Glossary, 2004]** H. Haas, and A. Brown (editors): Web Services Glossary, *W3C Working Group Note 11 February 2004*. Available from <http://www.w3.org/TR/ws-gloss/>.

**[WS-CDL, 2004]** N. Kavantzias, D. Burdett, and G. Ritzinger (editors): *Web Services Choreography Description Language Version 1.0. W3C Working Draft 17 December 2004*. Available from <http://www.w3.org/TR/ws-cdl-10/>.

**[Zaremba et al., 2005]** M. Zaremba, M. Moran, and T. Haselwanter: *WSMX Architecture. WSMO Working Draft v0.2, 2005*. Available from <http://www.wsmo.org/TR/d13/d13.4/v0.2/>.

## Appendix A: Human-Readable Syntax

### A.1. BNF-Style Grammar

### A.2. Example of Human-Readable Syntax

## Appendix B: Glossary

business process – collection of activities designed to produce specific outputs based on specific inputs;

state – a state is the complete set of properties describing a real world situation of an object at a given time;

role – participant in an interaction; the requestor or the provider of a service;

## Acknowledgement

The work is funded by the European Commission under the projects [DIP](#), [Knowledge Web](#), [InfraWebs](#), [SEKT](#), [SWWS](#), [ASG](#) and [Esperanto](#); by [Science Foundation Ireland](#) under the [DERI-Lion](#) project; and by the FIT-IT (Forschung, Innovation, Technologie - Informationstechnologie) under the projects [RW<sup>2</sup>](#) and [TSC](#).

The editors would like to thank to all the [members of the WSMX working group](#) for their advice and input into this document.