D13.9v0.1 WSMX Choreography

WSMX Working Draft 28 June 2005

Editors:
    Armin Haller
    James Scicluna
Authors:
    Armin Haller
    James Scicluna
    Thomas Haselwanter

This document is also available in a non-normative PDF version.

Table of contents

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 [Thatte, 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 [Kavantzas et al., 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 [Thatte, 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 interface description 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

The remainder of this document is structured as follows:

Chapter 2 gives an overview of the functionality of the WSMX Choreography component. It describes how the component fits to the overall architecture of WSMX, how the term Choreography in WSMO relates to its use in other specifications and defines the components interfaces. Chapter 3 defines the Choreography Interface specification syntax, starting with general WSML modeling elements, as well as syntax basics, such as the use of namespaces, non functional properties etc. Further the description part unique to the Choreography specification is defined and its underlying Abstract State Machine model is described. Finally the logical expressions allowed within this specification are delimited. Chapter 4 gives an overview of the components architecture with a description of the underlying Abstract State Machine engine. Further the execution semantics of the component are given. Chapter 5 provides an in-depth description of the implementation of the component itself. The object model is depicted as UML class diagrams and the chapter further includes a description of the applied rules engine. In chapter 6 we describe related work in the field of choreography specification languages and examine the design choice of Abstract State machine engines and rule-based systems. Finally chapter 7 concludes and outlines the direction of future work.

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 [Kavantzas et al., 2004] or in [Dijkman & Dumas, 2004]. In WSMO Choreography describes the behaviour of a service from one role instance. In [Dijkman & Dumas, 2004] this correspondents 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 distinct 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 [Kavantzas et al., 2004] or [Dijkman & Dumas, 2004]. In any other case the heterogeneities 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 specification 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.
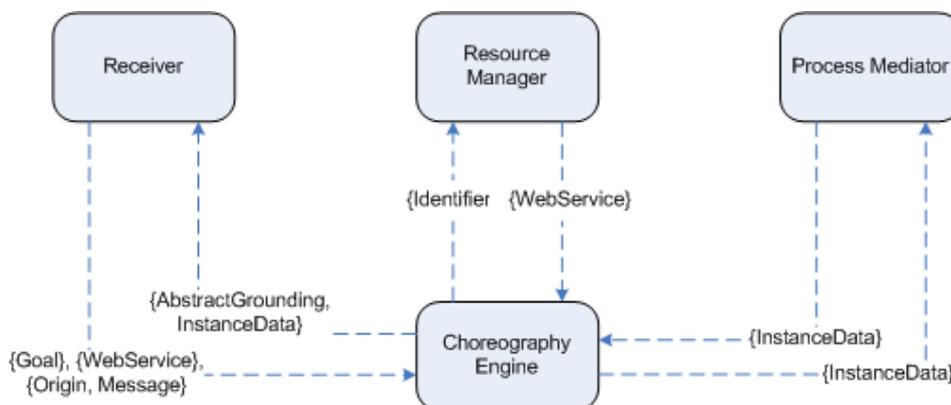

Figure 1: The Choreography Engine within the WSMX Context

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 WSML 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 WSML 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 WSML 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 WSML 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 WSML 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 WSML 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 heterogeneities. 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 ore 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.

In contrast to Abstract State Machines where a State *S* of a Vocabulary (signature) Σ is defined as a finite collection of function names, each with interpretations of these function names on a non-empty set *X* [Gurevich, 1995] in our model the signature Σ is defined by an ontological schema of the information interchanged in a Choreography Interface by specifying the used concepts, relations and functions of it. A state *S* of the signature $\Sigma = \Sigma(S)$ represents a stable status of the information space of the Choreography Interface defined by concrete attribute values of ontology instances and exists as long as these attribute values are not changed. Each of these instances are then bound to the mode denoting the operational behaviour of the respective state (see below).

Since a Choreography Interface specification describes the dynamic behaviour of a Service and represents an extension of a WSMO Webservice specification we can assume the required ontologies as already defined as they have been used to describe the capability of a Webservice. These ontologies used to represent states are subsequently imported in the Choreography Interface specification through the `importsOntology` keyword in the header of the Choreography Specification (see section 3.2.2).

Role

role	=	grounded_role
		| un_grounded_role
grounded_role	=	grounded_mode grounded_mode_list
un_grounded_role	=	un_grounded_mode un_grounded_mode_list

grounded_mode	=	'in'
		| 'out'
un_grounded_mode	=	'static'
		| 'controlled'
grounded_mode_list	=	grounded_mode_entry
		| grounded_mode_entry ',' grounded_mode_list
un_grounded_mode_list	=	id
		| id ',' un_grounded_mode_list
grounded_mode_entry	=	id 'withGrounding' idlist

Example:

```
in
   tr#route withGrounding _"http://example.com/#wsdl.interfaceRoute(RouteInterface/
provideRouteDetails)"

out
                    tr#reservation withGrounding _"http://example.com/#wsdl.
interfaceMessageReference(tripReservationInterface/bookTicket)"
```

The role block is used to define the communicative activities to be performed on a state. 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 communicative usage is denoted by the mode of a state, where we distinguish S_{in}, S_{out}, S_{stat}, S_{cont} where $S_{in} \subseteq S$; $S_{out} \subseteq S$; $S_{stat} \subseteq S$ and $S_{cont} \subseteq S$ whereas all are pairwise disjoint. These four state types denote the following [Roman et al., 2005b]:

- in - A in state S_{in} denotes 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. Instances of an ontology S_{in} can only be appear in the condition part of a rule.
- out - A out state S_{out} denotes 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. Instances of an ontology S_{out} can only appear in the update part of a rule.
- static - A static state S_{stat} where Sin subset S denotes 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. Instances of an ontology S_{stat} can only appear in the condition part of a rule.
- controlled - A controlled state S_{cont} denotes 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.

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

condition	=	expr ':-' expr
	|	expr

Example:

```
guardedTransition
  doForall {?route, ?start, ?end} with
    ?route[
    startLocation hasValue ?start,
    endLocation hasValue ?end
                       ] memberOf tr#route
                 and
    tr#connectionExists(?start,?end)
  then
    add(?reservation memberOf tr#reservation)
    add(?reservation[reservedRoute hasValue ?route]
             endForall
```

The set of transition rules in a Guarded Transition denote the state changes with regard to the evolution of the information space throughout the consumption of the functionality of the Webservice. The standard form of a guarded Transition *T* is:

if φ then *R* endif

where the Condition φ is a Boolean term and the update *R* is a set of transition rules. The update part defines all changes on the information space from an initial state *S* to *S'* where the condition φ is satisfied. These computation steps in an ASM in a given state consist in executing all updates of transition rules simultaneously whose condition is true, which yields to the next state if these updates are consistent [Börger & Stärk, 2003].

The `choose` and `doforall` constructs are introduced to comprise the extensions made to the basic paradigm of Abstract State Machines to handle nondeterministic behaviour in the case of the `choose` construct and parallel execution in the case of the `doforall` construct.

The following notation allows to express the simultaneous execution of a rule *R* for each variable *x* satisfying the condition φ.

doforall *x* with φ then *R* endforall

The `choose` construct is used to express non-determinism, where the rule *R* is executed with an arbitrary variable chosen from a list *x* among those satisfying the condition φ.

choose *x* with φ then *R* endchoose

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

The modifiers `add`, `delete` and `update` denote the actions performed on the knowledge base. The difference between

the add and `update` macro is that `update` deletes all old values, whereas `add` only adds a new statement to the knowledgebase.

As mentioned the `updaterule` block denotes the changes on the information performed in the transition *T* from *state S* to *state S'*. The execution of transitions is repeated until no state changes result, i.e. when two consecutive states are equal. In other words, the transitions are executed until a step occurs that does not produce any difference in state from the previous state. Updates are always performed simultaneously.

The following example shows how two updates are performed within the ASM.

```
if (?a[
  attr1 hasValue ?x
  ] memberOf tr#ontology
  and
  ?b memberOf tr#ontology[
  attr2 hasValue ?y])
then
  update(?a[attr1 hasValue ?x => ?y]) and update(?b[attr2 hasValue ?y => ?x])
endif
```

The guardedTransition shows that in contrast to most programming languages where we would need a temporary variable to store the value of ?x, we do not need a temporary variable here because the values are swapped simultaneously. For example, if ?x = 1 and ?y = 2, then the update statements ?x => ?y and ?y => ?x produces the update set (?x, 2), (?y, 1).

Due to the simultaneous behaviour of an ASM the order of updates within a step does not matter. However all of the updates in a step must be consistent. In other words, it is not allowed to have updates which contradict each other. For example, you can not update the attribute StartLocation to hasValue ?start and in the same block update it to hasValue ?end. This would produce a clash in the update set, since the ASM engine do not know which of the two updates should take effect. If updates do contradict, then they are called "inconsistent updates" and an error occurs [Börger & Stärk, 2003].

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

This section describes the architecture and behaviour of the Choreography Engine. The Choreography engine handles incoming messages from the other WSMX components but also manages instances of the ASM Engines of both the requester and provider. The ASM Engine can be considered as a component on its own and it is responsible to execute the ASM rules within the choreography interface descriptions of both the requester and the provider. The data transferred between these two entities, has to go through the Process Mediator in order to identify whether mediation is needed. In this section, the architecture of the core engine and the execution process are described. As far as the ASM engine is concerned, a minimalistic modular approach is taken into consideration. This would allow to reuse such a component within other WSMX component (such as an Orchestration or Process Mediator). The most basic elements of the choreography Engine are depicted in Figure 2 below.
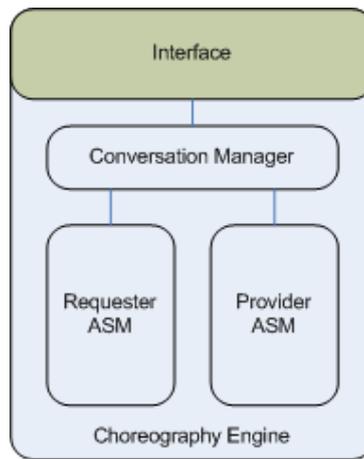
Figure 2: The basic elements of the Choreography Engine

The role of the Conversation Manager is two-fold. In the first place, if a request for loading a choreography is received, it must do so by making use of the Resource Manager. In the second place, it must determine to which ASM (requester or provider) the incoming messages belong to. The actual execution of the rules is handles by the ASM engines of the Requester and Provider choreographies.

4.1 ASM Engine

We introduce now an architecture for the ASM Engines which will be driving the choreography descriptions. Although the WSMO Choreography model differs from the pure ASM approach, the basic elements of both models are still the same. Our architecture is based on the approach taken in [Farahbod et al., 2005] which defines a core architecture for ASM execution. The diversities between ASM languages that stem from pure ASMs (such as ASML), has raised the need to specify a core engine that is capable to handle these different types of languages. The idea is to implement the minimal essential components which are able to run the most basic type of ASM. If some new model is to be introduced, then Plug-ins are used to specify both the syntax and the semantic information of the model.
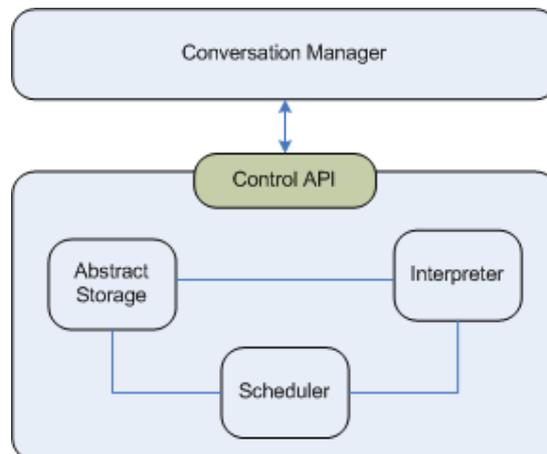

Figure 3: Main components of the ASM Engine

The components and operations in our Choreography Engine model are slightly different that the Core ASM approach. The Control API receives the notification from the Conversation Manager to execute the next step of the choreography description. In extension to this notification, the relevant instance data should be also produced. This is due to the fact that rules in the choreography fire after some instance data has been received. Also, a parser is not needed in this case since WSMX already parses the Web services when such descriptions are registered within the Resource Manager. Thus, an object model of the transition rules is readily available from such a component. The interpreter is responsible for executing the rules in the ASM by evaluating the condition and generating update sets. This will involve the usage of a back-end reasoner. The main responsibility of the abstract storage is to maintain a representation of the current state of the ASM. It enables to retrieve the values of instance data at a given state and apply the updates. The scheduler is responsible for orchestrating the execution steps of the ASM. The simplest operation of this component occurs during execution of sequential ASMs. After receiving a step notification from the Control API, it invokes the interpreter and

instructs the Abstract Storage to apply the updates if they are consistent. It then notifies the Control API of the changes that happened. For Multi-Agent (or Distributed) ASMs, the Scheduler uses some kind of policy to choose one Agent to execute (examples of such policies are Round-Robin and Priority based). The scheduler will then interact with the Abstract Storage to retrieve the selected set of agents. These agents are then assigned for execution and the updates are collected and performed via the Abstract Storage. The scheduler also manages inconsistencies of the update sets. In this case, we will assume that if inconsistent update sets occur, then the machine fails execution.

4.2 Execution Process

As already stated, at the moment it is assumed that the Choreography Engine has to deal with a one-to-one conversation, that is, one choreography for the requester and another one for the provider. It is expected that an orchestration in WSMO defines simpler choreographies which talk to the choreography interfaces of other Web services and hence the approach presented in this document can easily be integrated once orchestration in WSMO is better defined. We thereby describe here the execution process (together with the interaction) of the Choreography Engine's main elements (Figure 4). For the sake of clarity, the sequence diagram of the ASM Engines is illustrated separately in Figure 5.
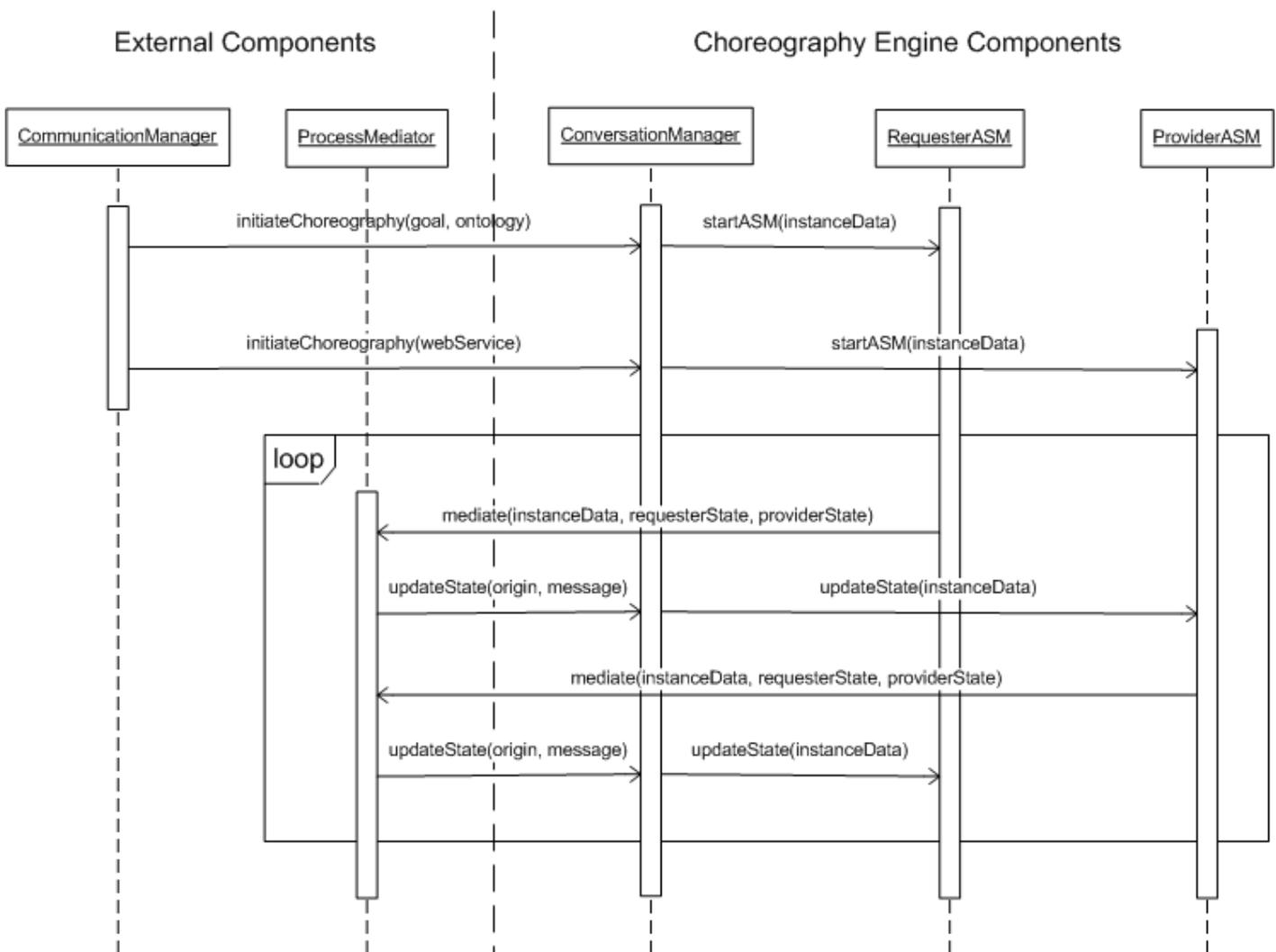

Figure 4: Execution Process of the Choreography Engine

The Communication Manager requests the Conversation Manager (the entry point to the Choreography Engine) to start the requester and provider choreographies. The Requester ASM is sent with all the instance data received from the client. The ASM of the provider may be initiated with no instance data since this will be received from the requester ASM. Once the provider ASM has been loaded, the requester sends the instance data to the Process Mediator. The latter determines whether some kind of mediation is needed and returns the instance data to the Conversation Manager accordingly. The Conversation Manager will then update the state of the Provider ASM from which the next instance data to be sent to the requester is received. Again, this goes through the Process Mediator and the (possibly) mediated

instance data is sent back to the Conversation Manager. This process is repeated until no further updates are possible on both sides.
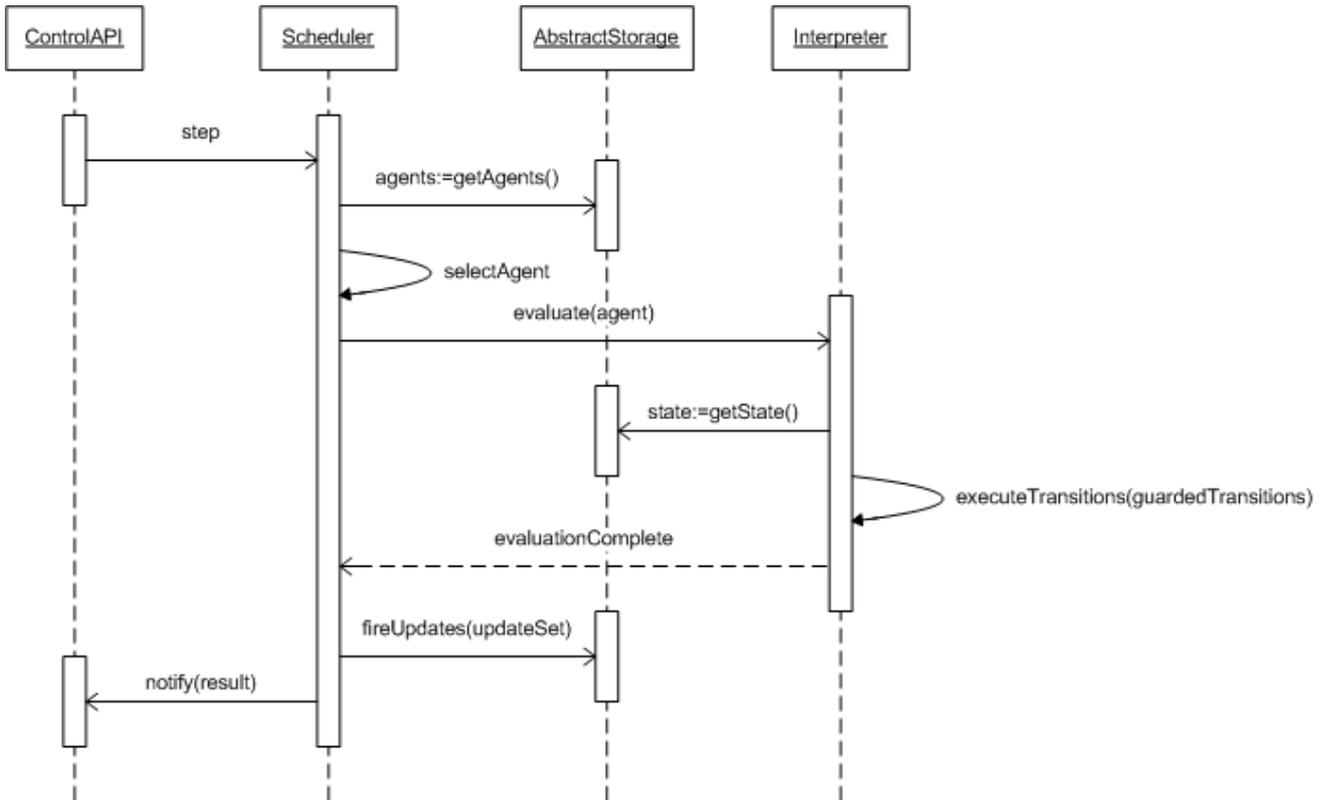

Figure 5: Functionality of the ASM Engine

Figure 5 above depicts the internal interaction between the components of the ASM Engine. Notice that the ControlAPI is the base class for both the Requester and Provider ASM components in Figure 4. The Control API instructs the Scheduler to perform a step in the machine. The scheduler retrieves the set of agents defined in the Abstract Storage and selects the appropriate agent to execute. Notice that since the choreography descriptions are based on a single agent, only one agent is selected. These operations have been kept for easy extensibility. After the agent has been selected, the Interpreter is notified to evaluate the program. The Interpreter retrieves the current state of the machine from the Abstract Storage and executes the Transitions Rules. Once executed, the Scheduler is notified and the update set is fired upon the Abstract Storage. Such update set (which is the result of the execution of the rules) is also notified to the environment via the Control API. Note that if an inconsistent update set is generated, then the machine stops execution and the environment is notified.

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

This section covers aspects about choreography and related tools. First an overview of existent choreography

languages is given. Section 6.1 covers both formal (e.g. OWL-S) and non-formal (e.g. WS-CDL) languages to define the dynamics of Web Service interfaces. Section 6.2 outlines the currently available ASM tools and finally a brief overview of Rule-Based systems is given (Section 6.3).

6.1 Web Service based Choreography and Orchestration Languages

WSDL [Christensen et al., 2001] allows to define a web service interface by means of the signatures of operations (their parameter types and return types) and its bindings. Hence it defines the static parameters how to invoke a Web Service, but not in what order to invoke the respective operations to achieve the desired functionality. Different languages have been proposed to extend this static interface description by providing the dynamics involved in calling Web Services.

WS-CDL [Kavantzas et al., 2004] is a non-executable business process description language building on top of WSDL with support to include and reference service definitions given in WSDL. As mentioned above it gives a global viewpoint of the common and complementary observable behavior between two or many roles. More specifically it is a declarative, XML based language that defines the information exchange that occurs and the jointly agreed ordering rules that need to be satisfied. Each party can then use this non-formal global definition to build a solution that conforms to the behaviour described in the global WS-CDL model.

The Business Process Execution Language for Web Services (BPEL4WS) [Thatte et al., 2003] is a language for modelling business processes serving two purposes; it offers the means to define both executable and abstract processes. Executable processes describe actual behaviour of a participant, comparable to the description of workflow processes of this participant. Abstract processes on the other hand describe the publicly visible message exchange between parties, without describing the internal behaviour behind this message exchanges. The second part is comparable to WS-CDL, describing the global observable behavior between parties involved in a collaboration.

OWL-S [OWL-S, 2004] is similar to WSMO an ontology framework for describing several aspects of web service. OWL-S provides means for describing: what a service provides and what it requires, how a service operates, and how a service is to be used. Every service can be seen as a process, described in a ProcessModel; there are three kinds of processes: atomic, simple and composite. An atomic process is directly invocable and executes in a single step - this means that the requester has no knowledge of how the service executes. A composite process is built from other (simple or composite) process using several control flow constructs. A simple process is an abstraction - it is not invocable, but can be conceived as a service executing as a single step. Although the process model of OWL-S follows the same intention than WSMO of describing both the client-service interaction for consuming a Web Service functionality and the composition of Web Services, there is no explicit separation of choreography and orchestration. The semantics of the process language have been provided using Petri-Nets [Narayanan & McIlraith, 2002].

6.2 ASM Tools

Most of the tools available for ASMs are not really meant for execution purposes. This is due to the fact that the primary purpose of ASMs is to allow formal specification, simulation and verification of systems. Hence, most of the tools outlined cannot be directly applied to Web Service scenarios since such a systems requires execution. However, there are other interesting applications in this respect, such as, simulation and verification of Web service behaviour, compatibility checks between Web service, deadlock freedom checking for the process model of a Web service and refinement of choreography interfaces with respect to the capability of the Web service (this particularly applies for WSMO).

The CoreASM project [Farahbod et al., 2005] is an initiative to implement the most basic core engine for an ASM. Our work presented here is based on the same approach with some differences. The CoreASM is composed of five main elements, namely, a ControlAPI, a Scheduler, an Interpreter and finally an Abstract Storage. The engine uses a plug-in system such that extensions to the basic ASM language can be easily used within the core. The implementation of such a system will be based on Java.

The KIV Tool is a system through which ASMs can be specified, verified and simulated in an interactive fashion through the use of a graphical user interface. It allows to use different specifications (e.g. algebraic specifications, dynamic logic and higher-order predicate logic). Proofs can be verified in a step by step fashion. The tool ships also with standard datatypes (such as for the Java language).

AsmGofer is an ASM programming system whose aim is to provide a modern ASM interpreter embedded in the Gofer functional programming language. AsmGofer introduced the notion of state and parallel updates into TkGofer (which

extends Gofer to support a graphical user interface). This language is not particularly suited for Web service applications. Also, since Gofer supports higher order functions than ASMs (which allow only first order functions and variables to be dynamic), the user has to specify manually the types. Furthermore, since AsmGofer is an interpreter it is usually slow which makes it unsuitable for Web service scenarios where sometimes performance is needed.

AsmL is an Abstract State Machine Language is a Microsoft proprietary language based on the .NET framework. The main objective of this language is to allow modeling, analysis, rapid prototyping and conformance checking of systems. AsmL is very similar to a programming language with intuitive and easy to learn constructs. However, it is not open-source and thus cannot be extended to suite other needs. Furthermore, plugging it to a Java based system (which is the base-code for WSMX) would hamper heavily the performance of WSMX.

6.3 Rule-Based Systems

It is worth putting a note about Rule-Based Systems in this context. Such systems share a lot of conceptual ideas with ASM implementation. A rule-based system is comprised of three main parts, namely, an Interpreter, a set of rules and a database. The knowledge is represented as facts stored in a database and as rules. The rules represent knowledge about a particular domain and the database represents the current state of the system which changes during execution. The interpreter (sometimes called the Expert System Shell) is fixed and is responsible to execute the rules upon the database. The state of an ASM may be encoded within the database and the guarded transitions as the rules. An example of a rule-based system is Jess. This system is encoded in Java and uses the Rete algorithm to process the rules. Jess is compatible with CLIPS but extends adds features like backward-chaining, working memory queries and the ability to manipulate and directly reason about Java objects. The language used for the scripts is however a lot different than WSML, thus requiring more effort to implement a translator.

7 Conclusion and Future Work

References

[Börger & Stärk, 2003] E. Börger, and R. Stärk: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer, Berlin et al. 2003.

[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/.

[Christensen et al., 2001] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana: Web Services Description Language (WSDL) 1.1. W3C Note v1.1, 2001. Available from http://www.w3.org/TR/wsdl.

[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.

[Farahbod et al., 2005] R. Farahbod, V. Gervasi, and U. Glaesser: An Extensible ASM Execution Engine. *In Proceedings of the 12th International Workshop on Abstract State Machines , Paris, March 2005:* pages 153-165.

[Gurevich, 1995] Y. Gurevich: Evolving Algebras 1993: Lipari Guide, Specification and Validation Methods, ed. E. Börger, Oxford University Press, 1995, 9--36.

[Kavantzas et al., 2004] N. Kavantzas, 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/.

[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/.

[Narayanan & McIlraith, 2002] S. Narayanan, A., S. McIlraith: Simulation, Verification and Automated Composition of Web Services. In *Proceedings of the 11th International World Wide Web Conference (WWW02)*, May, 2002.

[OWL-S, 2004] The OWL-S Coalition: OWL-S 1.1 Release, November 2004. Available from http://www.daml.org/services/owl-s/1.1/.

[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/.

[Thatte, 2003] Thatte, S. (editor): Business Process Execution Language for Web Services. Specification v1.1, 2003. Available from ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf.

[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/.

[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 – Business processes prescribe the way in which resources (e.g. information, humans, capital) of an enterprise are used, i.e. a collection of activities designed to produced 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. The main descriptions of a state are as an algebra in Abstract State Machines, as a propositional valuation (truth assignments to basic propositions as used in the propositional variants of temporal and dynamic logic), as an assignment to program variables as in the first-order variant of Dynamic Logic, as and algebra in Abstract State Machines or as a fully-fledged first order strtucture. In our model a state is defined by an ontological schema of the information interchanged in a Choreography Interface by specifying its used concepts, relations and functions.

role – participant in an interaction; the requestor or the provider of a service; In our model the role block is used to define the communicative activities to be performed on a state.

Acknowledgement