



D14v0.2. Ontology-based Choreography and Orchestration of WSMO Services

WSMO Final Draft 3rd February 2006

This version:

<http://www.wsmo.org/TR/d14/v0.2/20060203/>

Latest version:

<http://www.wsmo.org/TR/d14/v0.2/>

Previous version:

<http://www.wsmo.org/TR/d14/v0.2/20060113/>

Editors:

James Scicluna
Axel Polleres
Dumitru Roman

Co-Authors:

Dieter Fensel

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

Copyright © 2005 [DERI](#)®, All Rights Reserved. [DERI](#) liability, trademark, document use, and software licensing rules apply.

Table of contents

- [1. Introduction](#)
- [2. Overview of Abstract State Machines](#)
 - [2.1 Single-Agent ASM](#)
 - [2.1.1. Basic Terminology](#)
 - [2.1.2. Transition Rules](#)
 - [2.2 Multi-Agent ASM](#)
- [3. WSMO Choreography](#)
 - [3.1 State Signature](#)
 - [3.1.1 State](#)
 - [3.1.2 Roles of Concepts and Relations](#)
 - [3.2 Transition Rules](#)
- [4. WSMO Orchestration](#)
- [5. Future Work](#)
- [6. Conclusions](#)
- [Appendix A: Reference Syntax for WSMO Choreography](#)
- [References](#)
- [Acknowledgement](#)

1. Introduction

This document describes the Choreography and Orchestration elements of the interface definition of a WSMO Web service description [Roman et al., 2005]. These elements allow to describe the *behaviour* of the service from two orthogonal perspectives: *communication* (that is, how the service communicates with the client), and respectively *cooperation* (how the service uses other services or goals in order to fulfill its capability). Both views are separate abstractions of the actual implementation of the service.

The Choreography interface describes the behaviour of the service from the client's point of view; this definition is in accordance to the one given in the W3C Glossary [W3C Glossary, 2004]: *Web Services Choreography concerns the interactions of services with their users. Any user of a Web service, automated or otherwise, is a client of that service. These users may, in turn, may be other Web Services, applications or human beings.*

The Orchestration interface defines how the overall functionality of the service is achieved in terms of the cooperation with other services. It describes how the service works from the provider's perspective (i.e. how a service makes use of other services or goals in order to fulfill its capability). This complies with the W3C definition of Web Service Orchestration [W3C Working Group]: *An orchestration defines the sequence and conditions in which one Web Service invokes other Web Services in order to realize some useful function. That is, an orchestration is the pattern of interactions that a Web Service agent must follow in order to achieve its goal.*

The aim of this document is to provide a core conceptual model for describing choreography and orchestration interfaces in WSMO. The state-based mechanism for describing WSMO choreography and orchestration interfaces is based on the Abstract State Machine [Gurevich, 1993] methodology. An ASM is used to abstractly describe the behaviour of the service with respect to an invocation instance of a service. We have chosen an abstract machine model for the description of this interface since such a service invocation (e.g. the purchase of a book at amazon) may consist of a number of interaction steps. These interactions can be described by a stateful abstract machine.

ASMs have been chosen as the underlying model for the following three reasons:

- *Minimality*: ASMs provide a minimal set of modeling primitives, i.e., enforce minimal ontological commitments. Therefore, they do not introduce any ad-hoc elements that would be questionable to be included into a standard proposal.
- *Maximality*: ASMs are expressive enough to model any aspect around computation.
- *Formality*: ASMs provide a rigid framework to express dynamics.

The remainder of this document is organized as follows: [Section 2](#) provides an Overview of Abstract State Machines. [Section 3](#) provides a core conceptual model for WSMO choreographies, [Section 4](#) presents the conceptual model for WSMO orchestration interfaces, and finally [Section 5](#) and [Section 6](#) outline future work and draw some conclusive remarks.

2. Overview of Abstract State Machines

In this section, we give a brief overview of the main concepts of Abstract State Machines. Abstract State Machines (ASMs for short), formerly known as Evolving Algebras [Gurevich, 1995], provide means to describe systems in a precise manner using a semantically well founded mathematical notation. The core principles are the definition of ground models and the design of systems by refinements. Ground models define the requirements and operations of the system expressed in mathematical form. Refinements allow to express the classical divide and conquer methodology for system design in a precise notation which can be used for abstraction, validation and verification of the system at a given stage in the development process.

As described in [Börger & Stärk, 2003], Abstract State Machines are divided into two main categories, namely, Basic ASMs and Multi-Agent ASMs. The former express the behavior of a system within the environment. Multi-Agent ASMs allow to express the behavior of the system in terms of multiple entities that are collaborating to achieve a functionality. The latter can be further divided in two categories: Synchronous and Asynchronous Multi-Agent ASMs, both of which can be of a distributed or non-distributed nature. This classification is depicted in Figure 1 below.

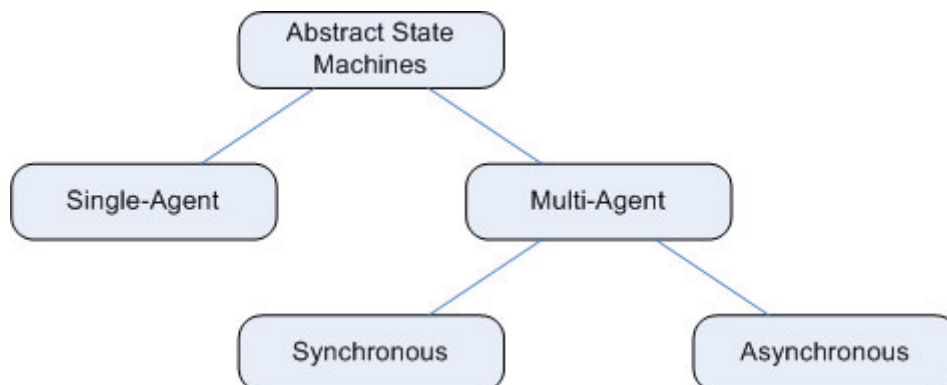


Figure 1: Different types of Abstract State Machines

2.1 Single-Agent ASM

A single-agent ASM (most commonly known as Basic ASM) is defined in terms of a finite set of transition rules which are executed in parallel. It may involve non-determinism as described below. We will first go through the basic definitions which will serve as the basis for the next sub-sections.

2.1.1. Basic Terminology

The signature Σ (also called a vocabulary) of an ASM is a finite collection of function names. Each function name f has an arity greater or equal to zero. Nullary function names are called *constants* and function names can be *static* or *dynamic* whereby the latter can be further classified as *in* (or *monitored*), *controlled*, *shared* (or *interaction*) or *out*. The arity of a function name determines the number of arguments that the function can take. Static functions never change during a run of a machine. Dynamic functions can be classified in four other categories, namely, *controlled*, *monitored* (or *in*), *interaction* (or *shared*) and *out*. Controlled functions are directly updatable by the rules of the machine M only. Thus, they can neither be read nor updated by the environment (described below). Monitored functions can only be updated by the environment and read by machine M and hence constitute the externally controlled part of the state. Shared functions can be read and updated by both the environment and the rules of the machine M . Out functions can be updated but not read by M , but can be read by the environment. Furthermore, ASMs define the so-called *derived* functions. There are functions neither updatable by the machine or the environment but which are defined in terms of other static and dynamic (and derived) functions.

The environment of an ASM consists of external agents interacting with the particular ASM. More precisely, the environment of an ASM is regarded as a set of variable assignments such that for a state S , a variable assignment for S is a finite function ζ which assigns elements of $|S|$ to a finite number of variables. $\zeta[x \rightarrow a]$ denotes a variable assignment which coincides with ζ except that it assigns the element a to the variable x :

$$\zeta[x \rightarrow a](y) = \begin{cases} a, & \text{if } y = x \\ \zeta(y) & \text{otherwise} \end{cases}$$

2.1.2. Transition Rules

The most basic rules are *Updates* which take the form of assignments (also called function updates) as follows:

$$f(t_1, \dots, t_n) := t$$

such that f is a function name with arity greater or equal to 0 and t_1, \dots, t_n are terms. For a signature Σ , terms are defined as follows:

1. Variables x, y, z, \dots are terms
2. Constants c in Σ are terms
3. If f is an n -ary function name in Σ , $n > 0$, and t_1, \dots, t_n are terms, then $f(t_1, \dots, t_n)$ is also a term

The execution of a set of such updates is carried out by changing the value of the occurring functions f at the indicated arguments to the indicated values in parallel. Hence, the parameters t_i and t are for example evaluated to the values v_i, v . The value of $f(v_1, \dots, v_n)$ is then updated to v which represents the value of $f(v_1, \dots, v_n)$ in the next state. The pairs of the function name f (specified by the signature) and the optional arguments (v_1, \dots, v_n) (which is a list of dynamic parameter values of any type), are called *locations*. These locations form the concept of the basic ASM object containers or memory units. The location-value pairs (loc, v) are called updates and represent a basic unit of state change in the ASM.

More complex transition rules are defined recursively, as follows. (Note that for the sake of clarity, we slightly deviate here from the original syntax used in [\[Börger & Stärk, 2003\]](#).) First, transition rules can be *guarded* by a *Condition* as follows:

if *Condition* **then** *Rules* **endif**

Here, the *Condition* is an arbitrary predicate logic formula without free variables which is evaluated to true or false. Such a guarded transition rule has the semantics that the *Rules* in its scope are executed in parallel, whenever the condition holds in the current state. Next, basic ASMs allow some form of universally quantified parallelism by transition rules of the form

forall *Variable* **with** *Condition* **do** *Rules*(*Variable*) **endforall**

The meaning of such rules is to execute simultaneously the enclosed *Rules* for each variable in *Variables* satisfying the *Condition* of the rule (where typically typically a variable will have some free occurrences in the *Rules* which are bound by the quantifier). Similarly, basic ASMs allow for non-deterministic choice by transition rules of the form

choose *Variable* **with** *Condition* **do** *Rules*(*Variable*) **endchoose**

Here, as opposed to the **forall** rule, one possible binding of the *Variable* such that the condition holds is picked non-deterministically by the machine and the *Rules* are executed in parallel only for this particular binding.

A single ASM execution step is summarized as follows:

1. Unfold (that is, evaluate the guards/conditions) the rules, according to the current state and conditions holding in that state, to a set of basic updates.
2. Execute simultaneously all the updates.
3. If the updates are consistent (i.e. no two different updates update the same location with different values, which means that there must not be a pair of updates $\{(loc, v), (loc, v')\}$ with $v \neq v'$), then the result of execution yields the next state.
4. All locations which are not affected by updates, keep their values.

These steps are repeated until no condition of any rule evaluates to true, i.e. the unfolding yields an empty update set. In case of inconsistent updates, the machine run is either terminated or an error is reported (or both).

2.2 Multi-Agent ASM

As described above, there are two types of Multi-Agent ASMs, namely, synchronous and asynchronous. A synchronous Multi-agent ASM consists of a set of basic ASMs each running their own rules and which are synchronized by an implicit global system clock. Such ASMs are equivalent to the set of all single-agent ASMs operating in the global state over the union of their state signatures. The global clock is considered as a step counter. Synchronous ASMs are particularly useful for analysing the interaction between components using precise interfaces over common locations. We consider this model insufficient for the description of the collaboration of Web services.

Asynchronous ASMs consist of a finite number of independent agents each executing a basic or structured ASM in its own local state. The problem which arises in such a scenario is that moves of the different agents cannot be compared due to different data, clocks and duration of execution. Furthermore, the global state is difficult to define since different agents may partially share the same state(s) or may not. The coherence condition for such ASMs is the *well-definedness* for a relevant portion of a state in which an agent is supposed to perform a step, thus providing the notion of "local" stable view of "the" state in which an agent makes a move.

3. WSMO Choreography

WSMO Choreography deals with interactions of the Web service from the client's perspective. We base the description of the behavior of a single service exposed to its client on the basic ASM model. WSMO Choreography interface descriptions inherit the core principles of such kind of ASMs, which summarized, are: (1) they are **state-based**, (2) they represents a state by a **signature**, and (3) it models state changes by **transition rules** that change the values of functions and relations defined by the signature of the algebra (which in our context is a non-empty set of ontologies).

In order to define the signature we use a WSMO ontology, i.e. definitions of concepts, their attributes, relations and axioms over these. Instead of dynamic changes of function values as represented by dynamic functions in ASMs we allow the dynamic modification of instances and attribute values in the state ontology. Note that the choreography interface describes the interaction with respect to a single instance of the choreography. The key extension compared with basic ASMs described above is that the machine signature is defined in terms of a WSMO ontology (possibly more than one) and the logical language used for expressing conditions is WSMML. This leads us to the notion of *Evolving Ontologies* (derived from the notion of *Evolving Algebras*, the initial name used for ASMs) since the Choreography ASM is in fact changing the values of concepts and relations within ontologies.

Taking the ASMs methodology as a starting point, a WSMO choreography consists of

three elements which are defined as follows:

Listing 1. WSMO choreography definition in the WSMO meta-model

```
Class choreography
  hasNonFunctionalProperties type nonFunctionalProperties
  hasStateSignature type stateSignature
  hasTransitionRules type transitionRules
```

Non-FunctionalProperties

Non-FunctionalProperties are the same as defined in [[Roman et al., 2005](#)] in Section 4.1.

State Signature

The State signature defines the state ontology used by the service together with the definition of the types of modes the concepts and relations may have.

Transition Rules

Transition rules that express changes of states by changing the set of instances.

The remainder of this section describes the main elements of the ASM-based choreography model. [Section 3.1](#) describes the state signature and [Section 3.2](#) describes the transition rules of the ASM.

3.1 State Signature

The signature of the machine is defined by (1) importing an ontology (possibly more than one) which defines the state signature over which the transition rules are executed, (2) an optional set of OO-Mediators if the imported state ontologies are heterogenous (3) a set of statements defining the modes of the concepts and relations and (4) a set of update functions. The default mode for concepts of the imported ontologies which are not listed explicitly in the modes statements is *static*. Note it is not allowed to assign the one of the modes *in* or *out* to concepts which have explicitly defined instance data in the imported ontologies by the state signature. Furthermore, since WSMML Full, Flight and Rule variants allow to define a concept and a relation with the same name, these elements must be preceded by the keyword *concept* or *relation* respectively as defined by the reference syntax in [Appendix A](#) of this document. If no keyword is specified, then it is assumed that the mode relates to a concept. Note also that here we don't define the grounding class. We assume that this is defined by some document reference by means of a URI (which may be specified within the mode declaration itself).

Listing 2. Definition of the State Signature in the WSMO meta-model

```
Class stateSignature
  hasNonFunctionalProperties type nonFunctionalProperties
  importsOntology type ontology
  usesMediator type ooMediator
  hasStatic type mode
  hasIn type mode
  hasOut type mode
  hasShared type mode
  hasControlled type mode

Class mode sub-Class {concept, relation}
  hasGrounding type grounding
```

3.1.1 State

The state for the given signature of a WSMO choreography is defined by all legal WSMO identifiers, concepts, relations and axioms. The elements that can change and that are used to express different states of a choreography, are instances of concepts and relations which are used similar to locations in ASMs. These changes are expressed in terms of creation of new instances or changes of attribute values.

3.1.2 Roles of Concepts and Relations

In a similar way to the classification of locations and functions in ASMs, the concepts and relations of an ontology are marked to support a particular role (or mode). These roles are of five different types:

- *static* - meaning that the extension of the concept cannot be changed. This is the default for all concepts and relations imported by the signature of the choreography.
- *in* - meaning that the extension of the concept or relation can only be changed by the environment. A **grounding** mechanism for this item may be provided that implements *write* access for the environment.
- *out* - meaning that the extension of the concept or relation can only be changed by the choreography execution. A **grounding** mechanism for this item must be provided that implements *read* access for the environment.
- *shared* - meaning that the extension of the concept or relation can be changed by the choreography execution and the environment. A **grounding** mechanism for this item may be provided that implements *read/write* access for the environment and the service.
- *controlled* - meaning that the extension of the concept is changed only by a choreography execution.

Since Web services deal with actual instance data, the classification inherits to instances of the respectively classified concepts and relations. That is, instances of *controlled* concepts and relations can only be created and modified by the choreography interface, instances of *in* concepts can only be read by the choreography, instances and *out* concepts can only be created by the choreography but not read or further modified after its creation. Instances of *shared* concepts and relations can be read and written by both the choreography and possibly the environment, i.e. can also be modified after creation. We suppose *shared* concepts are particularly important for groundings alternative to WSDL which do not rely on strict message passing such as semantically enables TupleSpaces (cf. [Fensel D., 2004](#)), in the future.

3.2 Transition Rules

As opposed to basic ASMs, the most basic form of rules are not assignments, but we deal with basic operations on instance data, such as adding, removing and updating instances to the signature ontology. To this end, we define atomic update functions to *add delete*, and *update* instances, which allow us to add and remove instances to/from concepts and relations and add and remove attribute values for particular instances. In WSMO Choreography, these basic updates are defined as a set of fact modifiers which are of four different types:

1. **add**(*fact*)
2. **delete**(*fact*)
3. **update**(*fact*^{*new*})
4. **update**(*fact*^{*old*} ⇒ *fact*^{*new*})

A *fact* can be either a membership *fact*(*x memberOf y*), an attribute *fact*(*x*[*a hasValue b*])

or $fact(r(t_1, \dots, t_n))$ for an n -ary relation. A combination in the form of a WSML molecule abbreviating conjunctions of membership and attribute facts (cf. [Bruijn et al., 2005]) is also allowed. The **add** modifier adds a new fact to the state unless it is already present. The **delete** modifier deletes a fact from the state, if present. For convenience, we also allow the use of **update** modifiers.

The third type of update rule can take the following forms:

- **update**($r(t_1, \dots, t_n)$) which is a shortcut for a combination of a forall and add as follows:
 - **forall**{ $?x_1, \dots, ?x_n$ } **with** ($?x_1 \neq t_1$ or ... or $?x_n \neq t_n$) **do delete**($r(?x_1, \dots, ?x_n)$) **endForall**
 - **add**($r(t_1, \dots, t_n)$)
- **update**(x **memberOf** y^{new}) which is a shortcut for a combination of a forall and add as follows:
 - **forall**{ $?y$ } **with** ($?y \neq y^{new}$) **do delete**(x **memberOf** $?y$) **endForall**
 - **add**(x **memberOf** y^{new})
- **update**(x [**a hasValue** b^{new}]) which is a shortcut for a combination of a forall and add as follows:
 - **forall**{ $?b$ } **with** ($?b \neq b^{new}$) **do delete**(x [**a hasValue** $?b$]) **endForall**
 - **add**(x [**a hasValue** b^{new}])

The fourth type of update rule can take the following forms:

- **update**($r(t_1, \dots, t_{i-1}, t_i^{old} \Rightarrow t_i^{new}, t_{i+1}, t_n)$) which is a shortcut for a combination of a delete and add as follows:
 - **delete**($r(t_1, \dots, t_{i-1}, t_i^{old}, t_{i+1}, t_n)$)
 - **add**($r(t_1, \dots, t_{i-1}, t_i^{new}, t_{i+1}, t_n)$)
- **update**(x **memberOf** $y^{old} \Rightarrow y^{new}$) which is a shortcut for a combination of a delete and add as follows:
 - **delete**(x **memberOf** y^{old})
 - **add**(x **memberOf** y^{new})
- **update**(x [**a hasValue** $b^{old} \Rightarrow b^{new}$]) which is a shortcut for a combination of a delete and add as follows:
 - **delete**(x [**a hasValue** b^{old}])
 - **add**(x [**a hasValue** b^{new}])

More complex transition rules are defined recursively, analogous to classical ASMs by **if-then**, **forall** and **choose** rules:

if *Condition* **then** *Rules* **endif**

forall *Variables* **with** *Condition* **do** *Rules* **endForall**

choose *Variables* **with** *Condition* **do** *Rules* **endChoose**

For unconditional non-determinism, we also allow to describe rules in the following form:

$rule_1 \mid rule_2 \mid \dots \mid rule_n$ which is a shortcut notation for:

```

choose {?x} with (?x=1 or ?x=2 or....or ?x=n) do
  if(?x=1) then rule1 endIf
  if(?x=2) then rule2 endIf
  ...
  if(?x=n) then rulen endIf
endChoose

```

This also implies that non-deterministic update rules are allowed.

4. WSMO Orchestration

It is envisioned that orchestration should make use of the Multi-Agent asynchronous ASM model to describe the interactions between Web services and Goals. These aspects are still to be further investigated and will be defined in future versions of this document.

As for the requirements of Orchestration Interfaces, it is planned by the authors to proceed as follows. The language will be based on the same ASMs model as Choreography interfaces which - in order to link to externally called services or (sub)goals that the service needs to invoke to fulfill its capability - needs to be extended as follows:

- Goals and Services can be used in place of rules, with the intuitive meaning that the respective goal/service is executed in parallel to other rules in the orchestration.
- The state signature defined in the choreography can be reused, i.e. external inputs and outputs of the service and the state of the choreography can be dereferenced also in the orchestration.
- Additionally the state signature for the orchestration interface can extend the state signature of the choreography interface, with additional *in/out/shared/controlled* concepts which need to be tied to the used services and rules by mediators
- Respective WW or WG mediators need to be in place to map the *in* and *out* concepts defined in the orchestration to the respective *out* and *in* concepts of the choreography interfaces in the used services and goals, i.e. these mediators state which output concepts are equivalent to which input of the called service/goal and vice versa, cf. Figure 2 below.

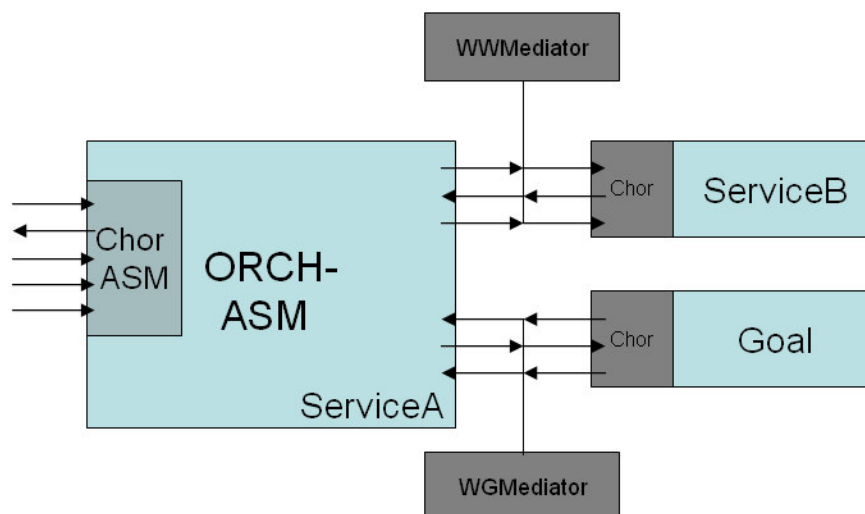


Figure 2: WSMO Orchestration

5. Future Work

Our current model of Ontology ASMs shall be further refined with future versions of WSML. A more readable high-level language with possibly graphical representation to ease modeling and tool support is on our agenda. Such language should be mappable to the ASM model defined in this document. This high-level language could be based on [UML Activity Diagrams](#) and work relating ASMs to this notation is already defined in [\[Börger & Stärk, 2003\]](#). Orchestration descriptions should address interactions with Goals and Web services using an asynchronous scenario with multi-agent ASMs.

6. Conclusions

This document presented a core conceptual model for modeling WSMO Choreographies and Orchestrations based on the ASM methodology. To this end we defined an ontology based ASM model for Single-Agent ASMs that are used as the base model for choreography interfaces. The state of these ontology ASMs machine are described by an ontology (or possibly a set of ontologies). The types of concepts are marked with in, out, shared, controlled and static defining the role of the particular concept in the machine. The interactions with a service can then be described with a set of transition rules. Multi-Agent ASMs shall be used as the model for describing interactions between different Web services in the orchestration in order to achieve the required functionality. For an example of WSMO Choreography, we refer the reader to ([\[Kopecky et. al., 2006\]](#)) which describes a Use Case for the Amazon E-Commerce Service.

Appendix A. Reference Syntax for WSMO Choreography

This appendix defines a reference syntax for choreography interfaces. To keep the document concise, the elements that are already defined in [\[De Bruijn et al., 2005\]](#) are not shown in the syntax.

Interfaces

An interface description in WSMO starts with the **interface** keyword and optionally followed by the identifier (as defined in [\[De Bruijn et al., 2005\]](#)) of the interface. What follows is an optional block of **nonFunctionalProperties**, an optional **importsOntology** statement and an optional **usesMediator** statement (these three elements are defined by the **header** block in [\[De Bruijn et al., 2005\]](#)). Finally, an optional **choreography** and **orchestration** block may be defined.

Similarly for an interface, a choreography block starts with an optional identifier followed by optional blocks of **nonFunctionalProperties**, **importsOntology** and **usesMediator** statements. Following these elements are the optional blocks of **stateSignature** and **transitions** containers. An orchestration simply starts with the **orchestration** keyword followed by an optional identifier. This element is yet to be better defined.

interface = 'interface' id? header* choreography? orchestration?

choreography = 'choreography' id? header* state signature? transitions?

orchestration = 'orchestration' id?

State Signature

A State Signature in a choreography description starts with the **stateSignature** keyword

followed by an optional identifier, an optional block of **nonFunctionalProperties**, an optional **importsOntology** statement and an optional **usesMediator** statement. Finally, the state signature defines an optional set of mode containers. Each **mode** container is defined by a mandatory mode name (**static**, **in**, **out**, **shared** or **controlled**) and a list of entries. Each entry may take a default form (which relates to a concept) an explicit concept definition or an explicit relation definition. The default form is defined by an **IRI** (as defined in [De Bruijn et al., 2005]) followed by an optional block of grounding IRIs. An explicit concept mode entry is defined by the keyword **concept** followed by an **IRI** followed by an optional block of grounding IRIs. The same applies for a relation mode entry except that instead of the keyword **concept**, **relation** is used. The grounding information is defined by the **withGrounding** keyword followed by a list of **IRIs**.

```

state signature = 'stateSignature' id? header* mode*
mode           = mode id mode entry list

mode entry list = {mode_entry} mode entry
                 | {mode_entry_list} mode entry', mode entry list

                 {static} 'static'
                 | {in} 'in'
mode id        = | {out} 'out'
                 | {shared} 'shared'
                 | {controlled} 'controlled'

                 {default_mode} iri grounding?
mode entry     = | {concept_mode} 'concept' iri grounding?
                 | {relation_mode} 'relation' iri grounding?

grounding      = 'withGrounding' grounding_info

grounding_info = {iri} iri
                 | {irilist} '{ irilist}'

irilist        = {iri} iri
                 | {irilist} iri', irilist

```

Transition Rules

Following the state signature block is the transition rules container. This is defined by the **transitionRules** keyword followed by an optional identifier, an optional **nonFunctionalProperties** block and a set of rule elements. A rule can take the form of an *if-then*, a *choose*, a *for-all*, a set of *pipelined-rules* (for non-determinism) or an *update-rule*. An *if-then* rule is defined by the **if** keyword, a WSML Logical Expression (condition), a **then** keyword, a non-empty set of rule elements and ending with the **endif** keyword. A *for-all* rule is defined by the **forall** keyword, a list of variable elements, a **with** keyword, a WSML Logical Expression (condition), a **do** keyword, a set of non-empty rule elements and ending with the **endforall** keyword. Similarly, a *choose* rule is defined by the **choose** keyword, a list of variable elements (as defined in [De Bruijn et al., 2005]), a **with** keyword, a WSML Logical Expression (condition), a **do** keyword, a set of non-empty rule elements and ending with the **endchoose** keyword. A pipelined rule is either a normal rule element or defined recursively by a rule followed by a pipe | followed by another set of pipelined rules.

```

transitions = 'transitionRules' id? nfp? rule*

```

```

    {if}      'if' condition'then' rule+ 'endlf'
    | {forall} 'forall' variablelist'with' condition'do' rule+ 'endforall'
rule      = | {choose} 'choose' variablelist'with' condition'do' rule+ 'endchoose'
    | {uncond} pipedReaders
    | {update} updaterule

condition = {restricted_le} expr

pipedReaders = {rule} rule
    | {pipedReader} rule'|' pipedReaders

```

An update rule is defined by a modifier keyword (**add**, **delete** or **update**), an open parenthesis (, a fact and a closing parenthesis). A fact can take the form of a preferred molecule, a non-preferred molecule or a fact molecule. The first form is defined by a term (as defined in [De Bruijn et al., 2005]), an optional attribute fact, a **memberOf** keyword, a term list (as defined in [De Bruijn et al., 2005]) and an optional fact update. An attribute fact is defined by an open square bracket [, an attribute fact list and a closing square bracket]. An attribute fact list can take the form of an attribute relation or a list of attribute relations (defined recursively) delimited by a comma. An attribute relation is defined by a term, a **hasValue** keyword, a term list and an optional fact update. A fact update is defined by an arrow of the form => followed by a term list. The non-preferred form of a fact is defined by a term, a **memberOf** keyword, a term list, an optional fact update and an attribute fact. A fact molecule is defined by a term and an attribute fact. A fact can also be related to a relation. Such a fact is defined by the '@' symbol (which clearly identifies that the modifier deals with an update of a relation) an identifier, an open parenthesis (, a list of term updates (depending on the number of arguments of the relation) and a closing parenthesis). The list of term updates is defined as a single term update or a term update followed by a comma followed by another term update. A term update is defined as a term, or as a term followed by the update symbol => and another new term.

```

updaterule = modifier(' fact')
    {add} 'add'
modifier = | {delete} 'delete'
    | {update} 'update'

    {fact_preferred} term attr fact? 'memberOf' termList fact update?
fact      = | {fact_nonpreferred} term'memberOf' termList fact update? attr fact
    | {fact_molecule} term attr fact
    | {fact_relation} '@' id(' term updates')

fact update = '=>' termList
attr fact = '[' attr fact list']'

attr fact list = {attr_relation} term'hasValue' termList fact update?
    | attr fact list',' term'hasValue' termList fact update?

term updates = {one_param} term update
    | {more_params} term update',' term updates

term update = {single} term
    | {move} [oldterm]: term'=>' [newterm]: term

```

new term = {new_term} '=>' term

References

[Börger, 1998] Egon Börger: "High Level System Design and Analysis Using Abstract State Machines", Proceedings of the International Workshop on Current Trends in Applied Formal Method: Applied Formal Methods, p.1-43, October 07-09, 1998

[Börger & Stärk, 2003] Egon Börger and Robert Stärk: "Abstract State Machines: A Method for High-Level System Design and Analysis", Springer-Verlag, 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 WSMO. *WSMO Final Draft v0.2*, 2005. Available from <http://www.wsmo.org/TR/d16/d16.1/v0.2/>.

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

[Fensel D., 2004] D. Fensel: Triple-space computing: Semantic Web Services based on persistent publication of information. Proc. of IFIP Int'l Conf. on Intelligence in Communication Systems 2004, Bangkok, Thailand, Nov 2004:43-53.

[Kopecky et. al., 2005]J. Kopecky, D. Roman (authors): WSMO Grounding, WSMO deliverable D24.2 version 0.1. available from <http://www.wsmo.org/2005/d24/d24.2/v0.1/>

[Kopecky et. al., 2006]J. Kopecky, D. Roman, J.Scicluna (eds.): WSMO Use Case: Amazon E-Commerce Service available from <http://www.wsmo.org/TR/d3/d3.4/v0.2/>

[Roman et al., 2005] D. Roman, H. Lausen, and U. Keller (eds.): Web Service Modeling Ontology (WSMO), WSMO deliverable D2 version 1.1. available from <http://www.wsmo.org/TR/d2/v1.2/>

[W3C Glossary, 2004] Hugo Haas, and Allen Brown (editors): Web Services Glossary, W3C Working Group Note 11 February 2004, available at <http://www.w3.org/TR/ws-gloss/>

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 RW² and TSC. The editors would like to thank to all the [members of the WSMO working group](#) for their advice and input into this document.



[webmaster](#) \$Date: 2006/02/03 17:03:15 \$
