



D14v0.3. Ontology-based Choreography of WSMO Services

WSMO Working Draft 21st April 2006

This version:

<http://www.wsmo.org/TR/d14/v0.3/20060421/>

Latest version:

<http://www.wsmo.org/TR/d14/v0.3/>

Previous version:

<http://www.wsmo.org/TR/d14/v0.3/20060331/>

Editor:

Dumitru Roman
James Scicluna

Co-Authors:

Dieter Fensel
Axel Polleres
Jos de Bruijn

Reviewers:

Stijn Heymans

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

Copyright © 2006 [DERI](#)®, All Rights Reserved. [DERI](#) liability, trademark, document use, and software licensing rules apply.

Table of contents

- [1. Introduction](#)
- [2. WSMO Choreography](#)
 - [2.1 State](#)

- [2.2 State Signature](#)
 - [2.3 Transition Rules](#)
 - [3. Syntax and Semantics](#)
 - [3.1 Syntax](#)
 - [3.1.1 State Signature](#)
 - [3.1.2 Transition Rules](#)
 - [3.2 Semantics](#)
 - [4. Future Work](#)
 - [5. Conclusions](#)
 - [Appendix A. Overview of Abstract State Machines](#)
 - [A.1 Single-Agent ASM](#)
 - [A.1.1. Basic Terminology](#)
 - [A.1.2. Transition Rules](#)
 - [2.2 Multi-Agent ASM](#)
 - [References](#)
 - [Acknowledgement](#)
-

1. Introduction

This document describes the Choreography element of the interface definition of a WSMO Web service description [[Roman et al., 2005](#)]. This element allows to describe the *behaviour* of the service from the *communication* perspectives (that is, how the service communicates with the client, in order to consume the functionality provided by 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 aim of this document is to provide a core conceptual model for describing choreography interfaces in WSMO, as well as providing a concrete syntax and semantics for this conceptual model. The state-based mechanism for describing WSMO choreography interfaces is inspired from 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 a core conceptual model for WSMO Choreographies, [Section 3](#) presents the syntax and semantics for a language that allows specifications conformant to the conceptual model introduced in Section 2, [Section 4](#) outlines possible future directions and [Section 5](#) concludes with some final remarks. [Appendix A](#) provides an overview of Abstract State Machines, to which the reader can refer to better understand the underlying starting point for the WSMO Choreography element.

2. 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 are defined by a **signature**, (3) the state is defined by ground facts, and (4) 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/ontologies. 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 WSML. 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 four elements which are defined as follows:

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

```

Class choreography
  hasNonFunctionalProperties type nonFunctionalProperties
  hasState type state
  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

The state of the Choreography is defined as a set of ground facts.

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 2.1](#) describes the state of the choreography, [Section 2.2](#) describes the state signature and [Section 2.3](#) describes the transition rules of the ASM.

2.1 State

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

2.2 State Signature

The signature of the machine is defined by (1) importing a non-empty set of ontologies which define 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*. Furthermore, since WSML 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 [Section 3.1](#) of this document. If no keyword is specified, then it is assumed that the mode relates to a concept. Note that the grounding class is not defined here in order to allow different types of grounding apart from WSDL.

```

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

```

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 unless defined otherwise in the state signature header.
- *in* - meaning that the extension of the concept or relation can only be changed by the environment and read by the choreography execution. 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 and read by the environment. 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 and read 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 and read only by the 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 execution, instances of *in* concepts can only be read by the choreography execution, instances of *out* concepts can only be created by the choreography execution 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.

2.3 Transition Rules

In a choreography specification *transition rules* express changes of states by changing the set of instances (adding, removing and updating instances to the signature ontology). The different transition rules allowed take the following form:

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

forall *Variables with Condition* **do** *Rules* **endforall**

choose *Variables with Condition* **do** *Rules* **endchoose**

The *Condition* (also called *guard*) under which a rule is applied is an arbitrary logical expression as defined by WSMML. The *Rules* may take the form of *Updates*, whose execution is to be understood as changing (or defining, if there was none) instances in an ontology. The update rules take the form of *add*, *delete*, and *update* instances; thus allowing to add and remove instances to/from concepts and relations and add and remove attribute values for particular instances. More complex transition rules can be defined recursively, analogous to classical ASMs by **if-then**, **forall** and **choose** rules:

Simultaneous execution provides a convenient way to abstract from irrelevant sequentiality and to make use of synchronous parallelism. The **forall** rule allows simultaneous execution of updates for each binding of a variable satisfying a given condition. The **choose** rule executes an update with an arbitrary binding of a variable chosen among those satisfying the selection condition. In this way, also non-deterministic update rules are allowed.

3. Syntax and Semantics

In this Section we define a reference syntax for choreography interfaces, in [Section 3.1 \[1\]](#), and we give the semantics of this syntax in [Section 3.2](#).

3.1 Syntax

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?

3.1.1 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

```

3.1.2 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+'endif'
    | {forall} 'forall' variablelist'with' condition'do' rule+'endforall'
rule         = | {choose} 'choose' variablelist'with' condition'do' rule+'endchoose'
    | {uncond} pipelined_rules
    | {update} updaterule
condition    = {restricted_le} expr
pipelined_rules = {rule} rule
    | {pipelined} rule'| pipelined_rules

```

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

```

3.2 Semantics

This section defines the semantics for WSMO Choreography descriptions. The semantics of a WSMO Choreography is defined in terms of the set of possible valid choreography runs which is associated with a choreography, given a set of facts, which can be seen as the starting point for the choreography.

The definition of the semantics of choreography descriptions is independent of the particular WSML variant which is used. In the following, whenever WSML is mentioned, we mean the particular WSML variant of the choreography description. Note that terms, formulae, entailment and satisfaction are defined differently for the different WSML variants, but all WSML variants have these notions. All the elements mentioned in this section are defined in [\[Bruijn et al., 2005\]](#) (unless explicitly defined here).

Ontology

The semantics of a WSMO Choreography is orthogonal to the semantics of the specific WSML variant. The variant is defined by the imported ontologies defined in the *importedOntologies* block in the state signature.

State

The state S of a WSMO Choreography is defined as a set of facts, where a *fact* is a ground atomic WSML formula.

A state S is said to be consistent with an ontology O iff $O \cup S$ is satisfiable, i.e., has a model according to the semantics of the specific WSML variant.

Update Rules

Update rules are either (i) primitive update rules of the form:

- **add**(a), or
- **delete**(a)

where a is a WSML atomic formula, which possibly includes free variables denoted as *parameter variables*, or (ii) non-primitive update rules of the form:

- **update**(a_{new}), or
- **update**($a_{old} \Rightarrow a_{new}$)

where a_{old} , a_{new} are WSML atomic formulae, which possibly include parameter variables. An update rule which contains no variables is a ground update rule.

Update Set

An update set is a set of primitive update rules without parameter variables. An update set U is said to be *consistent* if it does not contain the two elements **add**(a), **delete**(a) for the same fact a .

The *application* of an update set U to a state S yields a new state S^U such that:

$$S^U = S \setminus \{a \mid \text{delete}(a) \in U\} \cup \{a \mid \text{add}(a) \in U\}$$

Given an ontology O , a state S , and an update set U . U is said to be consistent with S wrt. O iff U is consistent and S^U is consistent with O .

Transition Rules

Transition rules are defined as follows:

1. Every ground update rule is a transition rule.
2. **if** φ **then** T **endif** is a transition rule.
3. **forall** V **with** ψ **do** T' **endforall** is a transition rule.
4. **choose** V **with** ψ **do** T' **endchoose** is a transition rule.

where

- φ is a formula with no free variables,
- V is a set of variables,
- ψ is a formula with possibly a number of free variables. These are interpreted as parameter variables and all free variables in ψ occur in V [3],
- T is a set of transition rules, and
- T' is a set of transition rules and/or non-ground update rules, where each variable which occurs in any non-ground update rule in T' , occurs also in V .

All types of transition rules can be defined within pipes to allow non-deterministic behaviour. Such transition rules take the form of:

$rule_1 \mid rule_2 \mid \dots \mid rule_n$ such that $rule_1 \dots rule_n$ are rules and $n > 1$ and are reduced to:

```
choose { $?x_1, ?x_2, \dots, ?x_n$ } with ( $?x_1 = 1$  or  $?x_2 = 2$  or... or  $?x_n = n$ ) do  
  if ( $?x_1=1$ ) then  $rule_1$  endif  
  if ( $?x_2=2$ ) then  $rule_2$  endif  
  ...  
  if ( $?x_n=n$ ) then  $rule_n$  endif  
endChoose
```

Transition rules of the forms 2-4 are also called *guarded* transition rules. Transition rules which include non-primitive update rules can be reduced to transition rules which contain only primitive update rules:

1. **update**($fact^{new}$) is translated as follows:

- **update**(α **memberOf** γ^{new}) is equivalent to a combination of a forall and add as follows:
 - **forall**{?y} **with** ($?y \neq \gamma^{new}$) **do delete**(α **memberOf** ?y) **endForall**
 - **add**(α **memberOf** γ^{new})
- **update**(α [β **hasValue** γ^{new}]) is translated as follows:
 - **forall**{?b} **with** ($?b \neq \gamma^{new}$) **do delete**(α [β **hasValue** ?b]) **endForall**
 - **add**(α [β **hasValue** γ^{new}])
- **update**($r(\sigma_1, \dots, \sigma_n)$) is translated as follows:
 - **forall**{ $?x_1, \dots, ?x_n$ } **with** ($?x_1 \neq \sigma_1$ or ... or $?x_n \neq \sigma_n$) **do delete**($\alpha(?x_1, \dots, ?x_n)$) **endForall**
 - **add**($r(\sigma_1, \dots, \sigma_n)$)

2. **update**($fact^{old} \Rightarrow fact^{new}$) is translated as follows:

- **update**(α **memberOf** $\gamma^{old} \Rightarrow \gamma^{new}$) is equivalent to a combination of a delete and add as follows:
 - **delete**(α **memberOf** γ^{old})
 - **add**(α **memberOf** γ^{new})
- **update**(α [β **hasValue** $\gamma^{old} \Rightarrow \gamma^{new}$]) is equivalent to a combination of a delete and add as follows:
 - **delete**(α [β **hasValue** γ^{old}])
 - **add**(α [β **hasValue** γ^{new}])
- **update**($r(\sigma_1, \dots, \sigma_{i-1}, \sigma_i^{old} \Rightarrow \sigma_i^{new}, \sigma_{i+1}, \sigma_n)$) is equivalent to a combination of a delete and add as follows:
 - **delete**($r(\sigma_1, \dots, \sigma_{i-1}, \sigma_i^{old}, \sigma_{i+1}, \sigma_n)$)
 - **add**($r(\sigma_1, \dots, \sigma_{i-1}, \sigma_i^{new}, \sigma_{i+1}, \sigma_n)$)

where $\alpha, \beta, \gamma, r, \sigma$ are identifiers as defined in [Brujin et al., 2005]. In the remainder we will assume (without loss of generality) that any transition rules does not include any non-primitive update rules.

Variable Substitution

Given a set of (parameter) variables $V = \{v_1, \dots, v_n\}$, a variable substitution θ for V is of the form $\theta = \{v_1 / t_1, \dots, v_n / t_n\}$ where t_i is an arbitrary ground WSMML term. The *application* of θ to some formula ϕ yields a formula $\phi\theta$ which is obtained from ϕ by replacing each unbound occurrence of v_1 with t_1 , each unbound occurrence of v_2 with t_2 , etc... The *application* of θ to a set of

transition and/or update rules T' yields a set of transition rules $T'\theta$ which is obtained from T' as follows: $T'\theta = \{\tau\theta \mid \tau \in T' \text{ and } \tau \text{ is an update rule}\} \cup \{\tau \mid \tau \in T' \text{ and } \tau \text{ is a guarded transition rule}\}$, where $\tau\theta$ is obtained from τ by replacing each unbound occurrence of v_1 with t_1 , each unbound occurrence of v_2 with t_2 , etc...

Choreography Interface

A Choreography Interface CI is defined as a 3-tuple, consisting of an Ontology O [\[2\]](#) (as defined in [\[Bruijn et al., 2005\]](#)), a set of transition rules T and a state S , that is:

$$CI = (O, T, S)$$

Associated Update Set

We say that an update set U is *associated* with a state S , given an ontology O and a set of transition rules T , if $U = \pi(T, O, S)$, with π defined as follows:

- $\pi(T, O, S) = \cup_{\tau \in T} \pi'(\tau, O, S)$
- $\pi'(\mathbf{add}(a), O, S) = \{\mathbf{add}(a)\}$
- $\pi'(\mathbf{delete}(a), O, S) = \{\mathbf{delete}(a)\}$
- $\pi'(\mathbf{if } \varphi \mathbf{ then } T, O, S) = \begin{cases} \pi(T, O, S) & \text{if } O \cup S \text{ entails } \varphi \\ \emptyset & \text{otherwise} \end{cases}$
- $\pi'(\mathbf{forAll } V \mathbf{ with } \psi \mathbf{ do } T \mathbf{ endForAll}, S) = \pi(\cup \{T\theta \mid \theta \text{ such that } \theta \text{ is a variable substitution for } V \text{ and } O \cup S \text{ entails } \psi\theta\}, O, S)$
- $\pi'(\mathbf{choose } V \mathbf{ with } \psi \mathbf{ do } T \mathbf{ endChoose}, O, S) = \begin{cases} \pi(T\theta, O, S) & \text{such that } \theta \text{ is a variable substitution for } V \text{ and } O \cup S \\ & \text{entails } \psi\theta \\ \emptyset & \text{otherwise} \end{cases}$

Choreography Interface Run

A choreography interface run ρ is defined as a sequence of states (S_0, \dots, S_n) .

Given a choreography interface $CI = (O, T, S)$ such that S is consistent with O , a choreography interface run $\rho = (S_0, \dots, S_n)$ is *valid* for CI iff

- $S_0 = S$,
- for $0 \leq i \leq n-1$,

- $S_i \neq S_{i+1}$,
- $U = \{\mathbf{add}(a) \mid a \in S_{i+1} \setminus S_i\} \cup \{\mathbf{delete}(a) \mid a \in S_i \setminus S_{i+1}\}$ is an update set associated with S_i , O and T ,
- S_{i+1} is consistent with O , and
- the run is *terminated*.

A run is terminated if either (a) there is an update set U associated with S_n , O and T such that U is not consistent with S_n wrt. O or (b) there is an update set U associated with S_n , O , and T such that $S_n = S_n^U$.

We denote a set of choreography interface runs as P . We denote the set of all valid choreography runs for a choreography interface CI as P^{CI} .

4. Future Work

Our current model of Ontology ASMs shall be further refined with future versions of WSML. A more 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 [\[Börger & Stärk, 2003\]](#).

5. Conclusions

This document presented a core conceptual model for modeling WSMO Choreographies 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. 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. Overview of Abstract State Machines

In this appendix, 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

A.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.

A.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 :

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

A.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 Variables with Condition do Rules(Variables) 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 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 Variables with Condition do Rules(Variables) 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).

A.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.

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 WSM. *WSML 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/>

Footnotes

[1] To keep the document concise, the elements that are already defined in [De Bruijn et al., 2005] are not shown in the syntax.

[2] Note that in case multiple ontologies are imported in a choreography, the ontology O is the union of the imported ontologies. As usual, the import process is recursive, i.e., O is the union of all imported ontologies, and, recursively, all ontologies imported by the imported ontologies.

[3] Note that we deviate here from the usual interpretation of free variables in WSML. In WSML, free variables are implicitly universally quantified over the entire formula. In conditions of **forall** and **choose** rules, free variables are interpreted extra-logically as parameters.

Acknowledgement

The work is funded by the European Commission under the projects DIP, Knowledge Web, InfraWebs, SEKT, SWWS, ASG and Esperonto; 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: 2006/04/20 07:25:42 \$
