1. Introduction

This document describes the Choreography and Orchestration parts of the interface definition of a WSMO service.
description [Roman et al., 2005]. These parts of a WSMO description describe the behavior of the service from two orthogonal perspectives: communication (how to communicate with the service from the client side such that the service will provide its capability), and respectively collaboration (how the service collaborates with other WSMO services or which goals it needs to resolve in order to achieve its capability). Both views are separate abstractions of the actual implementation of the service.

The Choreography part of a service interface describes the behavior of the service from a client point of view; this definition is in accordance to the following definition given by 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 part of a service interface defines how the overall functionality of the service is achieved in terms of the cooperation with other service providers with the actual implementation. It describes how the service works from the provider's perspective (i.e. how a service makes use of other WSMO services or goals in order to achieve 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 behavior 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 where these possible interactions can be abstractly 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, **Section 5** provides an example based on the Amazon Web service, and finally **Section 6** and **Section 7** outline future work and draw conclusions with some remarks.

## 2. Classical 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.
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.

2.1.1. Basic Transition Rules

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

\[ f(t_1, \ldots, t_n) := t \]

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 \( v_i \) and \( v \). The value of \( f(v_1, \ldots, v_n) \) is then updated to \( v \) which represents the value of the function \( f(v_1, \ldots, v_n) \) in the next state. The pair of the function name \( f \) (specified by the signature) and the optional arguments \( (v_1, \ldots, v_n) \) (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:

\[
\text{if Condition then Rules endIf}
\]

Here, the Condition is an arbitrary closed first order formula. 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

\[
\text{forAll Variable with Condition do Rules(Variable) endForAll}
\]

Here, the Variable is a variable occurring freely in Condition, with the meaning that the Rules[Variable/Value] are executed in parallel for all possible bindings of the Variable to a concrete Value such that the Condition[Variable/Value] holds in the current state. Here, Condition[Variable/Value] (and Rules[Variable/Value], resp.) stand for the condition (or rule, resp.) where each occurrence of Variable is replaced by Value.

Similarly, basic ASMs allow for non-deterministic choice by transition rules of the form

\[
\text{choose Variable with Condition do Rules(Variable) endChoose}
\]

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 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).

2.1.2. Function Classification

ASMs define a classification for functions that can be subject to updates or used in conditions. All functions are either static or dynamic. On the one hand, 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. 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.

As we will see later, we will base our model of the behavioral aspects of a single service on basic ASMs, operating on dynamic WSMO Ontologies, describing the state of the machine in terms of concepts, relations and their instances, where we can define the ontological axioms in terms of derived functions.

2.1.3. Control State ASMs

Readability and structure of general ASMs can be improved by introducing so called control states as syntactic sugar. Such control states allow to view ASMs as a straightforward extension of finite state machines and thus have desirable properties like high-level graphical representation and modularization of the machine. A Control State ASM is an ASM with one particular controlled function \( \text{ctl\_state} \) (which has as its range a finite number of integers or a finite enumeration of state-descriptors) and each transition rule having the form:

```plaintext
if \( \text{ctl\_state} = i \) then
  if Condi then
    Rulei
    \( \text{ctl\_state} := j_1 \)
  endif
  ...
  if Cond_n then
    Rulen
    \( \text{ctl\_state} := j_n \)
  endif
endif
```

2.1.4. Modularization

In order to structure and modularize ASM descriptions, it is allowed to define modules which take the following form:

```plaintext
ModuleName(Variable^1, ..., Variable^n) = \{Rules\}
```

In the definition of other rules these modules can be used as (possibly recursive) "rule calls" only when the parameters are instantiated by legal values (objects, functions, rules, so that the resulting rule has a well defined semantical meaning on the basis of the (informal) explanations given in the previous subsection. For details, cf. [Börger & Stärk, 2003]. Such modularization may be viewed as submachine calls which can also be recursive and nested, not not directly as composition of different (possibly distributed) ASMs since in a scenario with multiple ASMs run by different agents one need to consider details such as synchronous vs. asynchronous invocation, different clocks, etc. which is discussed in the following subsection.
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 represent 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.

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 based above is that the machine signature is defined in terms of a WSMO ontology and the logical language used for expressing conditions is WSML.

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

```xml
Class wsmoChoreography
    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) a set of statements defining the modes of the concepts (Section 3.1.1) and a set of update functions (Section 3.1.2). The default mode for concepts of the imported ontologies 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.
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.
- **controlled** - meaning that the extension of the concept is changed only by a choreography execution.
- **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.
- **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.
- **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.

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 of out concepts can only be created by the choreography but not read or further modified after its creation. Instances of shared concepts and relations are supposed to be read and written by both the choreography and possibly the environment, i.e. can also be modified after creation. We suppose shared concepts 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 three different types:

- **add** \((\text{fact})\)
- **delete** \((\text{fact})\)
- **update** \(\text{fact}_{old} \Rightarrow \text{fact}_{new}\), or simply
- **update** \(\text{fact}_{new}\)

A fact can be either a membership fact \((\text{memberOf})\), an attribute fact \((\text{hasValue})\) or a combination in the form of a WSML molecule abbreviating conjunctions of membership and attribute facts (cf. [Brujin et al., 2005]). 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. The update modifier in its first form marks the combination of deleting an old fact and adding a new one. The second form of update deletes all class membership or attribute values for a particular attribute and replaces these by the new fact. For a formal definition of the fact modifiers, please refer to Definition 3 of Appendix...
A.

REMARK: Known bug in this version: The updates only refer to class membership and attribute value facts for concepts, but there are no updates defined for relations.

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

if Condition then Rules endif

forAll Variables with Condition do Rules endForAll

choose Variables with Condition do Rules endChoose

Compared with basic ASMs, in WSMO choreography the following restrictions apply to Conditions and Variables:

- Variables are WSML variables as defined in [Bruin et al., 2005]
- A (WSML Full) Condition is a restricted form of WSML logical expressions where all free variables which are not bound be enclosing choose or forAll constructs are interpreted as being existentially quantified
- A WSML Core Condition is a WSML Full logical expression which consists only of molecules built up from memberOf and hasValue atoms and the logical connectives and or where all unbound variables are existentially quantified (i.e. a condition is a conjunctive query)

For a formal specification of the conditions, please refer to Definition 4 of Appendix A. We might extend the definition of conditions along future versions of WSML or as needed.

3.3 Relation of Transition Rules to Operations in WSDL grounding

REMARK: This subsection might better be moved to D24.2 WSMO Grounding.

Currently, one grounding mechanism to existing web services interfaces described in WSDL is defined in [Kopecky et. al., 2005]. The transition rules of WSMO Choreography descriptions, describe semantically the operations of the underlying WSDL which grounds the semantic description of the Web Service. However, there is no one to one correspondence between rules and operations in general, since in WSMO one can describe much more complex message exchange patterns than supported by WSDL. Instead of grounding rules to operations, the current WSDL grounding maps concepts to messages via their roles.

3.4 Relation of Transition Rules to Pre-conditions, Post-conditions and State invariants over service incovations

Further, also beyond WSDL, WSMO choreography interface descriptions allow to express arbitrary conditions over message exchanges. Pre-conditions over the input can be expressed in the if part of transition rules and arbitrary post-conditions can be attached via respective axioms in the signature ontology which are "triggered" by updates in the then part of a rule. Similarly, WSMO choreography allows to define state invariants (i.e. constraints over the states during a service invocation via its choreography interface via respective axioms in the signature ontology. Overall, WSMO choreography interfaces together with the attached signature ontologies allow you describe a more fine-grained behavioral description of the interactions with the service than the overall pre-conditions, post-conditions, assumptions and effects in the service capability.

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 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](image)

5. Example: Amazon Web service for Books

In this section we will present a Web service which enables searching and shopping cart management for books based on an existing Web service by amazon (Appendix B). This service allows the following operations:

- Searching by title, author(s), keyword(s) and price range
- Searching by a specific ASIN (Amazon Standard Identification Number)
- Cart Creation
- Get a particular cart
- Adding of items to the cart
- Clearing items off the cart

The terminology for semantic description of this service uses several ontologies, which we introduce and describe beforehand. We will first illustrate the general domain ontologies (commerce and books) and further define a task ontology for requests. Specific ontologies defined involve commerceRequests, amazonBooks and amazonRequests. Finally the semantic Web service description of Amazon is presented.

Listing 3 describes a simple book ontology. To keep our example concise, we will consider that this Amazon Web service handles only book-type items.

```xml
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-flight"

namespace {
    _"http://www.wsmo.org/ontologies/books#",
    loc _"http://www.wsmo.org/ontologies/location#",
    foaf _"http://xmlns.com/foaf/0.1/#",
    wsml _"http://www.wsmo.org/wsml/wsml-syntax#",
    dc _"http://purl.org/dc/elements/1.1#"
}

ontology _"http://www.wsmo.org/ontologies/books"

nonFunctionalProperties
    dc#title hasValue "Books Ontology"
    dc#creator hasValue "{James Scicluna}"
    dc#publisher hasValue "DERI Innsbruck"
```
Listing 4 describes a generic commerce ontology which defines the basic concepts related to products, items and carts.

```xml
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-flight"

namespace {
  _"http://www.wsmo.org/ontologies/commerce#",
  foaf _"http://xmlns.com/foaf/0.1/",
  wsml _"http://www.wsmo.org/wsml/wsml-syntax#",
  dc _"http://purl.org/dc/elements/1.1#"
}

ontology _"http://www.wsmo.org/ontologies/commerce"

nonFunctionalProperties
  dc#title hasValue "Commerce Ontology"
  dc#creator hasValue "Jos de Bruijn"
  dc#subject hasValue "E-commerce", "ok Buying"
  dc#description hasValue "A description of the basic domain of e-commerce (incomplete); t"
  dc#publisher hasValue "DERI Innsbruck"
  dc#contributor hasValue "James Scicluna, Jos de Bruijn"
  dc#date hasValue "$Date: 2005/10/09 02:48:17 "$
  dc#language hasValue "en-US"
  wsml#version hasValue "$Revision: 1.89 "$
endNonFunctionalProperties

concept price
  amount ofType (1) _decimal

concept product
  nonFunctionalProperties
    dc#description hasValue "A product for sale"
endNonFunctionalProperties

id ofType (0 1) _string
  title ofType _string
  price ofType price

concept cartItem
  nonFunctionalProperties
    dc#description hasValue "A cart item is described by a specific item and a quantity"
```
The task ontology is described in Listing 5 below. It defines a set of generic requests that can occur in an e-commerce scenario.

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-flight"
namespace {_"http://www.wsmo.org/ontologies/requests#",
foaf _"http://xmlns.com/foaf/0.1/",
wsml _"http://www.wsmo.org/wsml/wsml-syntax#",
dc _"http://purl.org/dc/elements/1.1#"}

ontology _"http://www.wsmo.org/ontologies/requests"

nonFunctionalProperties
dc#title hasValue "Requests Ontology"
dc#creator hasValue "{Jos de Bruijn}"
dc#subject hasValue "{Requests}"
dc#description hasValue "Requests which may be made to a service"
dc#publisher hasValue "DERI Innsbruck"
dc#contributor hasValue "{James Scicluna, Jos de Bruijn}"
dc#date hasValue "$Date: 2005/10/09 02:48:17 $"
dc#language hasValue "en-US"
wsml#version hasValue "$Revision: 1.89 $"
endNonFunctionalProperties

concept searchByKeyWords
nonFunctionalProperties
dc#description hasValue "Search by keywords"
endNonFunctionalProperties
keywords ofType _string

concept searchById
nonFunctionalProperties
dc#description hasValue "Search by identifier"
endNonFunctionalProperties
id ofType _string

concept addItem
The application ontology of amazon books is defined in Listing 6. The concepts extend the ones in the books ontology to support further features such as keywords and contact information.

```
nonFunctionalProperties
dc#description hasValue "Request to add"
endNonFunctionalProperties

concept deleteItem
nonFunctionalProperties
dc#description hasValue "Request to delete"
endNonFunctionalProperties
```

Listing 6. The amazon books application ontology

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-flight"

namespace { _"http://www.wsmo.org/ontologies/amazonbooks#",
books _"http://www.wsmo.org/ontologies/books#",
foaf _"http://xmlns.com/foaf/0.1/",
wsml _"http://www.wsmo.org/wsml/wsml-syntax#",
dc _"http://purl.org/dc/elements/1.1#"
}

ontology _"http://www.wsmo.org/ontologies/amazon/books"

nonFunctionalProperties
dc#title hasValue "Book Ontology"
dc#creator hasValue {"Jos de Bruijn"}
dc#subject hasValue {"Book", "Book Buying"}
dc#description hasValue "Describes the domain of books at Amazon"
dc#publisher hasValue "DERI Innsbruck"
dc#contributor hasValue {"James Scicluna, Jos de Bruijn"}
dc#date hasValue "$Date: 2005/10/09 02:48:17 $"
dc#language hasValue "en-US"
wsml#version hasValue "$Revision: 1.89 $"
endNonFunctionalProperties

importsOntology _"http://www.wsmo.org/ontologies/books"

concept amazonBook subConceptOf books#book
edition ofType (0 1) _integer
latestPrint ofType (0 1) _date
keywords ofType _string
website ofType _iri

concept amazonPublisher subConceptOf books#publisher
contactPhone ofType _string
contactEmail ofType _string
```

Listing 7 describes the commerce requests application ontology. It refines the concepts in the requests ontology to add features like price ranges when searching and specific requests related to carts.
Listing 7. Commerce requests application ontology

```xml
wsmlVariant "http://www.wsmo.org/wsml/wsml-syntax/wsml-flight"

namespace {
    "http://www.wsmo.org/ontologies/commercerequests#",
    requests "http://www.wsmo.org/ontologies/requests#",
    commerce "http://www.wsmo.org/ontologies/commerce#",
    foaf "http://xmlns.com/foaf/0.1/",
    wsml "http://www.wsmo.org/wsml/wsml-syntax#",
    dc "http://purl.org/dc/elements/1.1#"
}

ontology "http://www.wsmo.org/ontologies/commercerequests"

nonFunctionalProperties
dc#title hasValue "Requests Ontology"
dc#creator hasValue {"Jos de Bruijn"}
dc#subject hasValue {"Requests"}
dc#description hasValue "Requests in the area of e-commerce (incomplete)"
dc#publisher hasValue "DERI Innsbruck"
dc#contributor hasValue {"James Scicluna, Jos de Bruijn"}
dc#date hasValue "$Date: 2005/10/09 02:48:17 $"
dc#language hasValue "en-US"
wsml#version hasValue "$Revision: 1.89 $"
endNonFunctionalProperties

importsOntology {
    "http://www.wsmo.org/ontologies/requests",
    "http://www.wsmo.org/ontologies/commerce"
}

concept searchProduct subConceptOf requests#searchByKeywords
    nonFunctionalProperties
dc#description hasValue "A request to search for a product, based on different criteria"
dc#relation hasValue maxPriceGreaterThanMin
endNonFunctionalProperties
maxPrice ofType (0 1) commerce#price
minPrice ofType (0 1) commerce#price
title ofType _string

axiom maxPriceGreaterThanMin definedBy !-
    ?x memberOf searchProduct and
    (?x|maxPrice hasValue ?maxPrice] and
    ?x|minPrice hasValue ?minPrice] and
    ?maxPrice >= ?minPrice
).

custom concept getProductById subConceptOf requests#searchById
    nonFunctionalProperties
dc#description hasValue "A request to lookup a product using its id"
endNonFunctionalProperties

custom concept addToCart subConceptOf requests#addItem
    nonFunctionalProperties
dc#description hasValue "A request to add items to a cart"
endNonFunctionalProperties
cart ofType (1) commerce#cart
item impliesType (1 *) commerce#cartItem

concept deleteFromCart subConceptOf requests#deleteItem
    nonFunctionalProperties
dc#description hasValue "A request to delete items from a cart"
endNonFunctionalProperties
```

Listing 8 defines the specific requests related to the Amazon web service. It adds constraints on the search by keywords (whereby at least one keyword should be provided) and ID uniqueness of amazon carts.

Listing 8 . Requests ontology for Amazon

```xml
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-flight"

namespace {
  _"http://www.wsmo.org/ontologies/amazonrequests#",
  requests _"http://www.wsmo.org/ontologies/requests#",
  commerce _"http://www.wsmo.org/ontologies/commerce#",
  commercerequests _"http://www.wsmo.org/ontologies/commercerequests#",
  amazonbook _"http://www.wsmo.org/ontologies/amazonbooks#",
  foaf _"http://xmlns.com/foaf/0.1/",
  wsml _"http://www.wsmo.org/wsml/wsml-syntax#",
  dc _"http://purl.org/dc/elements/1.1#"
}

ontology _"http://www.wsmo.org/ontologies/amazonrequests"

nonFunctionalProperties
  dc#title hasValue "Amazon requests Ontology"
  dc#creator hasValue {"Jos de Bruijn"}
  dc#subject hasValue {"Requests"}
  dc#description hasValue "Requests for the Amazon web service (incomplete)"
  dc#publisher hasValue "DERI Innsbruck"
  dc#contributor hasValue {"James Scicluna, Jos de Bruijn"}
  dc#date hasValue "$Date: 2005/10/09 02:48:17 $"
  dc#language hasValue "en-US"
  wsml#version hasValue "$Revision: 1.89 $"
endNonFunctionalProperties

importsOntology {(_"http://www.wsmo.org/ontologies/commercerequests",
  _"http://www.wsmo.org/ontologies/amazon/books")

concept searchBooks subConceptOf commercerequests#searchProduct
  nonFunctionalProperties
    dc#description hasValue "A request to search for an item"
  endNonFunctionalProperties

concept item subConceptOf {amazonbook#amazonbook,commerce#product}
```
Listing 9 defines the Web service and its capability. For clarity reasons, the state signature and the transition rules of the choreography interface description are defined separately in Listing 10 and 11 respectively.

```xml
namespace { _"http://www.wsmo.org/webServices/amazonWS#",
  dc _"http://purl.org/dc/elements/1.1#",
  xsd _"http://www.w3.org/2001/XMLSchema#",
  amr _"http://www.wsmo.org/ontologies/amazonrequests#",
  commerce _"http://www.wsmo.org/ontologies/commerce#",
  comr _"http://www.wsmo.org/ontologies/commercerequests#",
  bk _"http://www.wsmo.org/ontologies/amazonbooks#",
  wsml _"http://www.wsmo.org/wsml/wsml-syntax#"
}

webService _"http://www.wsmo.org/webServices/amazonWS"
nonFunctionalProperties
dc#title hasValue "WSMO for Amazon"
dc#creator hasValue 
  ("James Scicluna","Axel Polleres")
dc#subject hasValue 
  ("Book", "Book Searching", "Book Buying")
dc#description hasValue "WSMO Web service for Book searching and ordering for amazon"
dc#publisher hasValue "DERI Innsbruck"
dc#contributor hasValue 
  ("James Scicluna","Jacek Kopecky", "Axel Polleres", "Jos de Bruij")
dc#date hasValue "$Date: 2005/10/09 02:48:17 $"
dc#type hasValue _"http://www.wsmo.org/2004/d2#webservice"
dc#format hasValue "text/html"
dc#language hasValue "en-US"
dc#relation hasValue 
```
"http://www.wsmo.org/ontologies/amazonrequests",
"http://www.wsmo.org/ontologies/commercerequests",
"http://www.wsmo.org/ontologies/amazonbooks#"
}

wsml#version hasValue "$Revision: 1.89 $"
wsml#endpointDescription hasValue "http://webservices.amazon.com/AWSECommerceSe
endNonFunctionalProperties

importsOntology {
  _"http://www.wsmo.org/ontologies/amazonrequests",
  _"http://www.wsmo.org/ontologies/commercerequests"
}

capability amazonWSCapability
  nonFunctionalProperties
    dc#description hasValue "The Amazon service offers:
    - Searching for Books with data related to title, author, keywords and price range
    - Lookup books with ASIN (Amazon Standard Identification Number)
    - Cart Management (Creation, Adding Items and Clearing)"
endNonFunctionalProperties

sharedVariables {?request}

precondition
  nonFunctionalProperties
    dc#description hasValue "The Amazon service handles requests for:
    - Searching request
    - Lookup request
    - Request to add items to a cart
    - request to clear the cart"
endNonFunctionalProperties

// REMARK: There is a problem here with sharedVariables, since we cannot assign whether universally or existentially quantified. We want to say here:
// "The precondition is that there exists a request."

definedBy
  //A search by author, keywords, title, minPrice or maxPrice
  ?request memberOf amr#searchBooks or
  //A search by ID
  ?request memberOf requests#searchByLd or
  //A request to create a cart
  ?request memberOf amr#createCart or
  //A request to get the cart
  ?request memberOf amr#getCart or
  //A request to add items to a cart
  ?request memberOf comr#addToCart or
  //A request to clear the cart
  ?request memberOf comr#cartClearRequest.

postcondition
  nonFunctionalProperties
    dc#description hasValue "The Service can return:
    - a list of searched items
    - a cart"
endNonFunctionalProperties
definedBy
  // 1) post-conditions for searchRequests:
  // General Problem: How is the container related to the request?
  // Shouldn't I have a possiblity to state that this container is a NEW container,
  // would I need the identifier of the request then?
  // The result for a search request by author is a container of products
// such that all items in the container are amazonBooks have the requested author:
(?request[author hasValue ?author] memberOf amr#searchBooks implies 
exists ?container (?container memberOf amr#itemContainer and 
forAll ?item (?container[items hasValue ?item] implies ?item[author hasValue ?author]) 
and

// The result for a search request by keyword is a container of products
// such that all items in the container are amzonBooks have the requested keyword:
(?request[keywords hasValue ?keyword] memberOf amr#searchBooks implies 
exists ?container (?container memberOf amr#itemContainer and 
forAll ?item (?container[items hasValue ?item] implies ?item[keywords hasValue ?keyword]) 
and

// The result for a search request by title is a container of products
// such that all items in the container have the requested title (not necessarily only books)
(?request[title hasValue ?title] memberOf amr#searchBooks implies 
exists ?container (?container memberOf amr#itemContainer and 
forAll ?item (?container[items hasValue ?item] implies ?item[title hasValue ?title]) 
and

// The result for a search request by minPrice is a container of products
// such that all items in the container have a price >= than the requested minPrice (not n:
(?request[minPrice hasValue ?minPrice] memberOf amr#searchBooks implies 
exists ?container (?container memberOf amr#itemContainer and 
forAll (?item,?price) (?container[items hasValue ?item] and ?item[price hasValue ?price 
?price >= ?minPrice)) 
and

// The result for a search request by maxPrice is a container of products
// such that all items in the container have a price =< than the requested maxPrice (not n:
(?request[maxPrice hasValue ?maxPrice] memberOf amr#searchBooks implies 
exists ?container (?container memberOf amr#itemContainer and 
forAll (?item,?price) (?container[items hasValue ?item] and ?item[price hasValue ?price 
?price =< ?maxPrice)) 
and

// The result for a search request by id is a container of products
// such that all items in the container have the requested id:
(?request[id hasValue ?id] memberOf requests#searchById implies 
exists ?container (?container memberOf amr#itemContainer and 
forall (?item) (?container[items hasValue ?item] implies ?item[id hasValue ?id])) 
and

// 2) PostConditions for other requests:
// Note: It is not obvious how to express that the cart created is related to this request
// in any way, nor that the cart-id did not exist before.
// How can I express now "with a new ID"?
(?request[items hasValue ?cartItem] memberOf comr#addToCart implies 
exists ?cart (?cart[items hasValue ?cartItem] memberOf amr#amazonCart) 
and

// What is the postcondition of a getCart-request?
// it has no effect except that the content of that cart is returned, but no cart is created.
// In its current form this post-condition is void (a tautology).
(?request[id hasValue ?id] memberOf amr#getCart and ?cart[id hasValue ?id] implies 
true /* To be completed, currently missing: */ 
) 
and

//A request to add items to a cart.
// Note: Isn't this incomplete? Actually, shouldn't the request say is that
// for already existing items the amount in the cart is increased? Otherwise, if you have al
// one item X in the cart and you request to add one more, the postcondition does not g
// the additional item is added!!!!
(?request[id hasValue ?id, item hasValue ?item] memberOf comr#addToCart and 
?cart[id hasValue ?id] memberOf amr#amazonCart implies 
?cart[item hasValue ?item] 
) 
and
// A request to remove an item from the cart:
// Note: similar problem as before: Should the amount given for the item be subtracted
// or is simply the whole item removed? In the former case we would need to dereference
// of the amount for the respective item. We proceed simpler now by just asserting that the
(?request[id hasValue ?id, item hasValue ?item] memberOf comr#deleteFromCart and
?cart[id hasValue ?id] memberOf amr#amazonCart implies
neg ?cart[item hasValue ?item])
// A request to clear the cart:
(?request[id hasValue ?id] memberOf comr#cartClearRequest and
?cart[id hasValue ?id] memberOf amr#amazonCart implies
neg exists ?item (?cart[item hasValue ?item])
).

Listing 10. State signature of the interface buyWithAmazon
choreography
stateSignature
importsOntology{
  _"http://www.wsmo.org/ontologies/commerce",
  _"http://www.wsmo.org/ontologies/amazonrequests",
  _"http://www.wsmo.org/ontologies/commercerequests"
}

in
  _"http://webservices.amazon.com/AWSECommerceService/2005-03-23#wsdl.interface"
},
amr#amazonCart withGrounding {
  _"http://webservices.amazon.com/AWSECommerceService/2005-03-23#wsdl.interface"
}

out
commerce#itemContainer withGrounding {
  _"http://webservices.amazon.com/AWSECommerceService/2005-03-23#wsdl.interface"
},
amr#amazonCart withGrounding {
  _"http://webservices.amazon.com/AWSECommerceService/2005-03-23#wsdl.interface"
}

The transitions are described in Listing 11 and are expressed as forAll-do since the web service handles multiple
instances from different requests.
Listing 11: Transition Rules of the Choreography for Amazon Web service

```java
transitionRules
/*
 * Search by keywords, author, title, price range and create a list of items. The
 * condition checks also that such an item with the specified attribute values
 * exists in the state.
 */
forAll {?request} with
  (?request[
    keywords hasValue ?keywords,
    author hasValue ?author,
    title hasValue ?title,
    minPrice hasValue ?minPrice,
    maxPrice hasValue ?maxPrice
  ] memberOf amr#searchBooks and
  exists (?book[
    keywords hasValue ?keywords,
    author hasValue ?author,
    title hasValue ?title,
    minPrice hasValue ?minPrice,
    maxPrice hasValue ?maxPrice
  ] memberOf bk#amazonBook )
  ) do
  add(_# memberOf commerce#itemContainer)
endForall

/*
 * Lookup with an ASIN and return a list of items
 * PROBLEM: same as for the searchWithAmazon interface
 */
forAll {?request, ?item} with
  ((?request[
    id hasValue ?id
  ] memberOf amr#searchById) and
  exists (?item[
    id hasValue ?id
  ] memberOf commerce#product) )
  do
  add(_#[
    items hasValue ?item
  ] memberOf amr#itemContainer)
endForall

/*
 * Creation of a Cart
 */
forAll {?request, ?cartItems} with
  (?request[
    items hasValue ?cartItems
  ] memberOf commerce#cartItemContainer) do
  add(_#[
    items hasValue ?cartItems
  ] memberOf amr#amazonCart)
endForall

/*
 * Get Cart. The condition checks also the existance of a cart
 * in the knowledge base.
 */
forAll {?cartGet} with
  (?cartGet[
    id hasValue ?cartId
  ] memberOf amr#getCart and
  exists (?cart[
    id hasValue ?cartId
  ] memberOf amr#amazonCart)
```
6. Future Work

Our current model of Ontology ASMs shall be further refined with future versions of WSML. The formal semantics of rules in our ontology ASM model which is currently defined in the Appendix needs further refinement and further proof-of-concept descriptions. A more readable high-level language with possibly graphical representation to ease modeling and tool support is on our agenda. Such language should be mapplicable to the ASM model defined in this document. Such a 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.

7. Conclusions

This document presented a core conceptual model for modeling WSMO Choreographies and Orchestrations based on the ASMs 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.
Appendix A. Definitions

We hereby present the proposed syntax for choreography interface descriptions. Notice that some of the structures specified in this syntax have not yet been presented in the main document.

Definition 1: Basic WSMO Choreography

The basic WSMO Choreography is defined by a set of Non-Functional Properties as defined in [De Bruijn et al., 2005] Section 2.2.3, a signature $\Sigma$ and a set of transition rules $Rule$.

$$\text{choreography} = \text{'choreography'} \text{id}? \text{header}\ast \text{state} \_\text{signature}\ast \text{transitions}\ast$$

Definition 2: State Signature

The state signature $\delta$ is defined by a set of WSMO state ontologies $O$ and a set of Role definitions. The set of ontologies define the state on which the conditions of the transition rules are evaluated and also onto which the updates are performed. The Role definitions mark concepts with in, out, shared, controlled or static which are "inherited" by variables and instances which are a member of the particular concept.

$$\text{state} \_\text{signature} = \text{'stateSignature'} \text{iri}? \text{header}\ast \text{role}$$

The state signature description is identified by an IRI followed by a header and role declarations. The state ontology (or ontologies) are declared via the importsOntology statement as part of the header. Roles are of two main categories, namely, grounded and un-grounded. The former are of type in, out and shared and the rest are static and controlled as defined below.

$$\text{role} = \{\text{grounded} \_\text{role}\} \text{grounded} \_\text{role}$$
$$\text{grounded} \_\text{role} = \text{grounded} \_\text{mode} \text{grounded} \_\text{mode} \_\text{list}$$
$$\text{un} \_\text{grounded} \_\text{role} = \text{un} \_\text{grounded} \_\text{mode} \text{un} \_\text{grounded} \_\text{mode} \_\text{list}$$
$$\text{grounded} \_\text{mode} = \{\text{in}\} \text{'in'}$$
$$\text{un} \_\text{grounded} \_\text{mode} = \{\text{static}\} \text{'static'}$$

$$\text{grounded} \_\text{mode} \_\text{list} = \{\text{grounded} \_\text{mode} \_\text{entry}\} \text{grounded} \_\text{mode} \_\text{entry}$$
$$\text{grounded} \_\text{mode} \_\text{list} = \{\text{grounded} \_\text{mode} \_\text{entry}\} \text{grounded} \_\text{mode} \_\text{entry}, \text{grounded} \_\text{mode} \_\text{list}$$
$$\text{grounded} \_\text{mode} \_\text{entry} = \text{id} \text{'withGrounding'} \text{id} \text{list}$$
$$\text{un} \_\text{grounded} \_\text{mode} \_\text{list} = \{\text{un} \_\text{grounded} \_\text{mode} \_\text{entry}\} \text{id}$$
$$\text{un} \_\text{grounded} \_\text{mode} \_\text{list} = \{\text{un} \_\text{grounded} \_\text{mode} \_\text{entry}\} \text{id}, \text{un} \_\text{grounded} \_\text{mode} \_\text{list}$$

Definition 3: Rule

The current grammar presents three types of rules, namely, if-then, forAll-do, choose-do and update-rule.

$$\text{transitions} = \text{'transitionRules'} \text{iri}? \text{rule}\ast$$
$$\text{rule} = \{\text{if}\} \text{'if condition'then'} \text{rule}\ast \text{'endIf'}$$
$$\{\text{choose}\} \text{'choose'} \text{variablelist}'with' \text{condition}'do' \text{rule}\ast \text{'endChoose'}$$
$$\{\text{forAll}\} \text{'forAll variablelist'}'with' \text{condition}'do' \text{rule}\ast \text{'endForAll'}$$
$$\text{updaterule}$$

The rules "if", "choose" and "forAll" allow to specify nested rules whereas update-rules are atomic. There are three kinds of atomic updates, namely, add, delete and update:
Facts can be of two different natures, namely membership facts (a memberOf b) and attribute facts (a[attr hasValue val]). It is allowed also to specify multiple facts in a single modifier. In such case, there are two ways of specifying a membership fact which includes attribute facts, namely, a memberOf b[attr_fact_list] or a[attr_fact_list] memberOf b, the latter being the preferred one.

Definition 4: Conditions in basic WSMO Choreography

Conditions in the basic WSMO choreography are syntactically derived form of WSML Logical Expressions as defined in [De Bruijn et al., 2005]. More precisely, we restrict WSML Logical Expression by disallowing the use of naf, !- , and :-.

Semantically, we slightly deviate from the definition of WSML logical expressions as in Section 2.8 of [De Bruijn et al., 2005] with respect to the scope of variables:

The scope of a variable in a condition is always defined by its quantification. If a variable is neither quantified inside the condition nor bound by an enclosing forall or choose rule, the variable is implicitly existentially quantified outside the condition.

The intuitive reason why we close off free variables is that conditions can be viewed as queries over the state ontology: When checking such a condition over the state we are not interested in proving that it holds for all possible bindings of a free variable, but only whether such a binding exists.

Formal Semantics of Choreography Interfaces

The intuitive semantics of the Ontology ASMs defined in this document is as follows. When beginning to interact with a service via its choreography interface you start with the initial state and all possible interaction sequences with the service are defined by the possible runs of the Ontology ASM defined in the Choreography interface.

Definition (initial state): The initial state of a Choreography Interface (i.e. at the time when interaction with the service starts, is defined by the state signature. That is, the initial state is the set $S_0$ consisting of all instances of all concepts and relations defined in the imported ontologies.

A run of an Choreography interface is defined analogously to the runs of basic ASMs in [Börger & Stärk, 2003], i.e. runs are defined as sequences of possible single execution steps:

Possible executions steps are defined by

$S' = S \setminus \{fact | delete(fact \in U) \} \cup \{fact | add(fact \in U) \}$

where $S$ is the current state, $U$ is a consistent update set, and $S'$ is the resulting state of applying $U$ in $S$.

We recall that the choreography interface is defined by a set of transition rules $R$: Let $O$ denote the imported signature ontology(ies). We define update sets for $U_{<R,S>}$ inductively:

- $U_{<add(fact),S>} = add(fact)$
\begin{itemize}
  \item \(U_{\text{delete}(\text{fact}),S} = \text{delete}(\text{fact})\)
  \item \(U_{\text{update}(\text{a memberOf } A \Rightarrow B),S} = \{\text{delete}(\text{a memberOf } A),\text{add}(\text{a memberOf } B)\}\)
  \item \(U_{\text{update}(\text{a[att} \text{hasValue } A \Rightarrow B)],S} = \{\text{delete}(\text{a[att} \text{hasValue } A]),\text{add}(\text{a[att} \text{hasValue } B])\}\}
  \item \(U_{R,S} = \bigcup_{r \in R} U_{r,S}\)
  \item \(U_{\text{if } \text{Cond} \text{ then } R,S} = \)
    \begin{itemize}
      \item \(U_{R,S}\) if \(O \cup S\) entails \(\text{Cond}\)
      \item \(\emptyset\) otherwise
    \end{itemize}
  \item \(U_{\text{forall } ?\text{Var with } \text{Cond do } R \text{ endForAll},S} = \{U_{R\theta,S} \mid \theta\text{ such that }\theta(?)\text{Var} / id\text{ is an identifier such that }O \cup S\text{ entails }\text{Cond}\theta\}\)
  \item \(U_{\text{choose } ?\text{Var with } \text{Cond do } R \text{ endChoose},S} = \)
    \begin{itemize}
      \item \(U_{R\theta,S}\) where \(\theta(?)\text{Var} / id\text{ is a non-deterministically chosen identifier such that }O \cup S\text{ entails }\text{Cond}\theta\)
      \item \(\emptyset\) if \(O \cup S \cup \{\exists ?\text{Var} (\text{Cond})\}\) is unsatisfiable.
    \end{itemize}
\end{itemize}

An update set \(U\) is consistent if it does not contain any two elements \(\text{add}(\text{fact})\) and \(\text{delete}(\text{fact})\) and the resulting state \(S' = S \setminus \{\text{fact} \mid \text{delete}(\text{fact})\} \cup \{\text{fact} \mid \text{add}(\text{fact})\}\) is consistent with the signature ontology, i.e. \(S' \cup O\) is satisfiable.

A run of an ASM stops if the update set induced by the rules above is empty, i.e. no more rules can fire.

### Appendix B. Amazon Web Service for Books

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/
xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://webservices.amazon.com/AWSECommerceService/2005-03-23">
  <types>
      xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:complexType name="ItemSearchRequest">
        <xs:sequence>
          <xs:element name="Author" type="xs:string" minOccurs="0"/>
          <xs:element name="Keywords" type="xs:string" minOccurs="0"/>
          <xs:element name="MaximumPrice" type="xs:nonNegativeInteger" minOccurs="0"/>
          <xs:element name="MinimumPrice" type="xs:nonNegativeInteger" minOccurs="0"/>
          <xs:element name="Title" type="xs:string" minOccurs="0"/>
        </xs:sequence>
      </xs:complexType>
    </xs:schema>
      xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:complexType name="ItemLookupRequest">
        <xs:sequence>
          <xs:element name="ItemId" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
    </xs:schema>
      xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:complexType name="CartGetRequest">
        <xs:sequence>
          <xs:element name="CartId" type="xs:string" minOccurs="0"/>
        </xs:sequence>
      </xs:complexType>
    </xs:schema>
      xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:complexType name="CartAddRequest">
        <xs:sequence>
          <xs:element name="CartId" type="xs:string" minOccurs="0"/>
        </xs:sequence>
      </xs:complexType>
    </xs:schema>
      xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:complexType name="CartRemoveRequest">
        <xs:sequence>
          <xs:element name="CartId" type="xs:string" minOccurs="0"/>
        </xs:sequence>
      </xs:complexType>
    </xs:schema>
      xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:complexType name="CartClearRequest">
        <xs:sequence>
          <xs:element name="CartId" type="xs:string" minOccurs="0"/>
        </xs:sequence>
      </xs:complexType>
    </xs:schema>
      xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:complexType name="CartItemRequest">
        <xs:sequence>
          <xs:element name="ItemId" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
    </xs:schema>
  </types>
</definitions>
```
<xs:sequence>
  <xs:element name="ASIN" type="xs:string" minOccurs="0"/>
  <xs:element name="Quantity" type="xs:positiveInteger" minOccurs="0"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:complexType>
  <xs:sequence>
    <xs:element ref="tns:Cart" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:element name="CartClearResponse">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="tns:Cart" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="Items">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="tns:Item" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="Cart">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="CartId" type="xs:string"/>
      <xs:element ref="tns:CartItems" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="Item">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="ASIN" type="xs:string"/>
      <xs:element name="MediumImage" type="tns:Image" minOccurs="0"/>
      <xs:element ref="tns:CustomerReviews" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="CartItems">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="SubTotal" type="tns:Price" minOccurs="0"/>
      <xs:element name="CartItem" type="tns:CartItem" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:complexType name="CartItem">
  <xs:sequence>
    <xs:element name="CartItemId" type="xs:string"/>
    <xs:element name="ASIN" type="xs:string" minOccurs="0"/>
    <xs:element name="Quantity" type="xs:string"/>
    <xs:element name="Title" type="xs:string" minOccurs="0"/>
    <xs:element name="Price" type="tns:Price" minOccurs="0"/>
    <xs:element name="ItemTotal" type="tns:Price" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Price">
  <xs:sequence>
    <xs:element name="Amount" type="xs:integer" minOccurs="0"/>
    <xs:element name="CurrencyCode" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

<xs:element name="ItemAttributes">
  <xs:complexType>
    <xs:sequence>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<x:schema>
<x:element name="Author" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
<x:element name="ISBN" type="xs:string" minOccurs="0"/>
<x:element name="ListPrice" type="tns:Price" minOccurs="0"/>
<x:element name="Title" type="xs:string" minOccurs="0"/>
</x:sequence>
</x:complexType>
</xs:element>
</xs:schema>
<x:message name="ItemSearchRequestMsg">
  <part name="body" element="tns:ItemSearch"/>
</message>
<x:message name="ItemSearchResponseMsg">
  <part name="body" element="tns:ItemSearchResponse"/>
</message>
<x:message name="ItemLookupRequestMsg">
  <part name="body" element="tns:ItemLookup"/>
</message>
<x:message name="ItemLookupResponseMsg">
  <part name="body" element="tns:ItemLookupResponse"/>
</message>
<x:message name="CartGetRequestMsg">
  <part name="body" element="tns:CartGet"/>
</message>
<x:message name="CartGetResponseMsg">
  <part name="body" element="tns:CartGetResponse"/>
</message>
<x:message name="CartAddRequestMsg">
  <part name="body" element="tns:CartAdd"/>
</message>
<x:message name="CartAddResponseMsg">
  <part name="body" element="tns:CartAddResponse"/>
</message>
<x:message name="CartCreateRequestMsg">
  <part name="body" element="tns:CartCreate"/>
</message>
<x:message name="CartCreateResponseMsg">
  <part name="body" element="tns:CartCreateResponse"/>
</message>
<x:message name="CartClearRequestMsg">
  <part name="body" element="tns:CartClear"/>
</message>
<x:message name="CartClearResponseMsg">
  <part name="body" element="tns:CartClearResponse"/>
</message>
<portType name="AWSECommerceServicePortType">
  <operation name="ItemSearch">
    <input message="tns:ItemSearchRequestMsg"/>
    <output message="tns:ItemSearchResponseMsg"/>
  </operation>
  <operation name="ItemLookup">
    <input message="tns:ItemLookupRequestMsg"/>
    <output message="tns:ItemLookupResponseMsg"/>
  </operation>
  <operation name="CartGet">
    <input message="tns:CartGetRequestMsg"/>
    <output message="tns:CartGetResponseMsg"/>
  </operation>
  <operation name="CartAdd">
    <input message="tns:CartAddRequestMsg"/>
    <output message="tns:CartAddResponseMsg"/>
  </operation>
  <operation name="CartCreate">
    <input message="tns:CartCreateRequestMsg"/>
<output message="tns:CartCreateResponseMsg"/>
</operation>
<operation name="CartClear">
  <input message="tns:CartClearRequestMsg"/>
  <output message="tns:CartClearResponseMsg"/>
</operation>
</portType>
<binding name="AWSECommerceServiceBinding" type="tns:AWSECommerceServicePortType">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="ItemSearch">
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
  <operation name="ItemLookup">
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
  <operation name="CartGet">
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
  <operation name="CartCreate">
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
  <operation name="CartAdd">
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
  <operation name="CartClear">
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>
<service name="AWSECommerceService">
    <port name="AWSECommerceServicePort" binding="tns:AWSECommerceServiceBinding">
        <soap:address location="http://soap.amazon.com/onca/soap?Service=AWSECommerceServicePort"/>
    </port>
</service>
</definitions>

References


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: 2005/10/09 02:48:17 $