WSMX Deliverable

D13.2 v0.2
WSMX Execution Semantics

WSMX Working Draft – 25th April 2005

Authors:
Maciej Zaremba and Eyal Oren

Editors:
Maciej Zaremba

This version:
http://www.wsmo.org/TR/d13/d13.2/v0.2/20050425/
Latest version:
http://www.wsmo.org/TR/d13/d13.2/
Previous version:
http://www.wsmo.org/TR/d13/d13.2/v0.2/20050404/

Copyright © 2005 DERI ®, All Rights Reserved.
DERI liability, trademark, document use, and software licensing rules apply.

1 Introduction

The Web Services Execution Environment (WSMX) is an execution environment for dynamic data and process mediation, discovery and invocation of Web Services. WSMX uses WSMO to describe all aspects related to this data and process mediation, discovery and invocation.

The goal of WSMX is to provide both a testbed for WSMO and to show the viability of using WSMO to achieve dynamic interoperable Web Services. The complete work on WSMX includes defining the conceptual model, defining an execution semantics for the environment, describing an architectural and software design and building a working implementation.

WSMX is based on event-driven architecture composed of loosely coupled components. This kind of architecture facilitates the creation of execution semantics for a system since the activities of the components are stimulated by events as they occur and there are no fixed bindings between components. Components can create or consume events but they cannot invoke each other directly (strictly speaking, only core component can interact with other components).

In this deliverable we define the execution semantics of WSMX. For readers unfamiliar with the term execution semantics, we first describe what is meant by this concept and why we model the execution semantics. Next we explore different modelling techniques for defining execution semantics in the existing literature and choose an appropriate modelling technique. Using this technique we present four mandatory WSMX execution semantics (each WSMX instance has to provide them), so that its operational behavior is formally and unambiguously specified. We are aware that there will be a demand for defining execution semantics for WSMX by third parties. There is a separate section characterizing this issue.

2 Methodology

2.1 What is 'execution semantics'?

Execution semantics, or operational semantics, is the formal definition of the operational behavior of a system. It describes in a formal, unambiguous language how the system behaves. Because the meaning of the system (to the outside world) consists of its behavior, this formal definition is called 'execution semantics'.

2.2 Why are we modelling execution semantics

WSMX has two functions in the complete body of WSMO: it serves both as a testbed for WSMO and as an example of its implementation. WSMX can be used as an example implementation to demonstrate the viability of WSMO framework or as a reference for others who want to build their own WSMO based execution environment. Unlike the WSMO work it is clearly not prescriptive. It does not tell others how to build a WSMO execution environment. In that sense, the execution semantics described here are strictly part of the design process of WSMX. Its meaning and relevance should be found in improving either the design process or the result of this process, the actual software.

A perfect design process (not just software design) should result in a design that is both an adequate response to the user's requirements and a feasible

directory for the implementor who will build the end result. A design therefore serves two purposes, both to guide the builder in the work of building the system, and to certify that what will be built will satisfy the user's requirements.

2.3 How do we model execution semantics

Several methods exist to model software behaviour. Some of them model system behaviour in less precise way like UML diagrams, other impose more formal requirements on model like Petri Net based methods. These methods have different characteristics: some are more expressive than others, some are more suited for a certain problem domain than others. Some methods are graphical like UML or Petri Net, some are based on logic terms like Fuzzy logic; some methods have tool support for modelling, for verification, for simulation or for automatic code generation and others don't.

To choose a method, we first define our requirements: first of all, the method should be as expressive as needed to define the behaviour of the software. Then, as outlined before, since the main advantage of using formal methods lies in improving the developers' understanding of the system, the resulting model should be easily understandable and unambiguous in its meaning. Thirdly, the method should allow verification of certain interesting properties of the modelled system. Lastly, since some methods are better suited for modelling a certain problem domain than others; the method should be suitable for modelling our specific problem domain [5].

3 Formal model of execution semantics

3.1 Modelling technique

We will describe the execution semantics using classical Petri nets. The Tool we are using, CPNTools [6], make it possible to model so-called high-level Petri-nets [1], extending classical Petri nets with hierarchy, colour and time.

Petri net is a directed graph with nodes represented either by places or by transitions. Places are mostly depicted as circles and transitions are depicted as rectangles. These two parts of graph occur in alternate manner. Places can hold tokens what denote an active state and when all places directed to particular transition are in a active state transition can be executed ("fired" in terms of Petri net glossary). Transition can carry out some elaborate task and task can be forwarded for completion to the external world (e.g. application or user). According to transition result new discrete number of tokens can be created and routed to directly outgoing places from completed transition.

The hierarchy feature, makes it possible to decompose a transition into so-called subnets, which allows us to break down a large model into smaller pieces and to model incrementally. Timed Petri nets introduce the notion of a global time, in which every token gets a certain time-stamp; this makes it possible to describe time duration of transitions.

Coloured Petri nets are classical Petri nets extended with the notion of identity. The addition of colour means basically that tokens can be distinguished from each other, for instance by giving every token a certain type and value. Since every token now has a certain value, every transition gives its output tokens certain values; or to put it another way, a transition now becomes a relation between the value of the input tokens and the value of the output

tokens. In addition, a transition can state conditions over its input tokens, that must be satisfied before the transition can become enabled.

Extending Petri nets with hierarchy and colour does not improve the expressivity of a Petri nets, but it does make them more concise and readable.

3.2 Model checking and simulation

The tool we are using makes it possible to verify certain properties of the model; it can check some simple properties such as syntactical correctness, unreachable (unused) places, or unsatisfiable conditions. The tool also allows for complex analysis of the constructed model, using state-space analysis (which is basically an exhaustive search through all possible states of the model).

The tool also allows for simulation of the constructed model. This simulation is very useful, since it greatly enhances the modeller's understanding of the system. When a modeller is not completely familiar with Petri nets, it is quite easy to construct a Petri net that does not follow the modeller's intention. When running a simulation, these errors will easily be detected.

The simulation also helps greatly in understanding the system's functionality, and in discussing the model with others. In that sense, simulation serves as an abstract prototype of the future system. It can be used to test and analyse the modelled system in detail. To allow for simulation, our model is available for download[1].

4 WSMX mandatory execution semantics

Four mandatory execution semantics are needed for each instance of WSMX system. Any one of the predefined execution semantic can be selected in response to a particular WSML message. The WSML message contents determinate the way the system behaves. Since WSMX is an event driven system, its behavior is specified by the order of events. Events exchange is conducted via Tuple Space, that provides persistent shared space enabling seamless interaction between components without direct events exchange between them. Interaction is carried out by exploiting a publish-subscribe mechanism. Execution Semantics are started by the Execution Manager (i.e. it initiates exchange of event via Tuple Space according to a selected formal specification). WSMX works with the WSMO conceptual model and with the wsmo4j [2] data model that is compliant with the WSMO specification v1.0.

Each WSMX execution semantic follows the path depicted on Figure 1. However not all steps are mandatory. If the service requester can understand WSML then it is not necessary to use an Adapter component for conversion purposes.

The execution semantics of WSMX follows the component-based paradigm. This means that the execution semantics of the complete system treats components as 'black-boxes' - we do not model decisions that take place inside those components. For a complete model of the behaviour of the system the execution semantics of the components needs to be taken into account. Modelling the execution semantics of the individual components is, however, the responsibility

[1] all models created in CPN Tools are available from http://www.wsmo.org/TR/d13/d13.2/v0.2/20050404/wsmxcpn.zip; the tool itself can be downloaded from http://wiki.daimi.au.dk/cpntools/

[2] http://wsmo4j.sourceforge.net

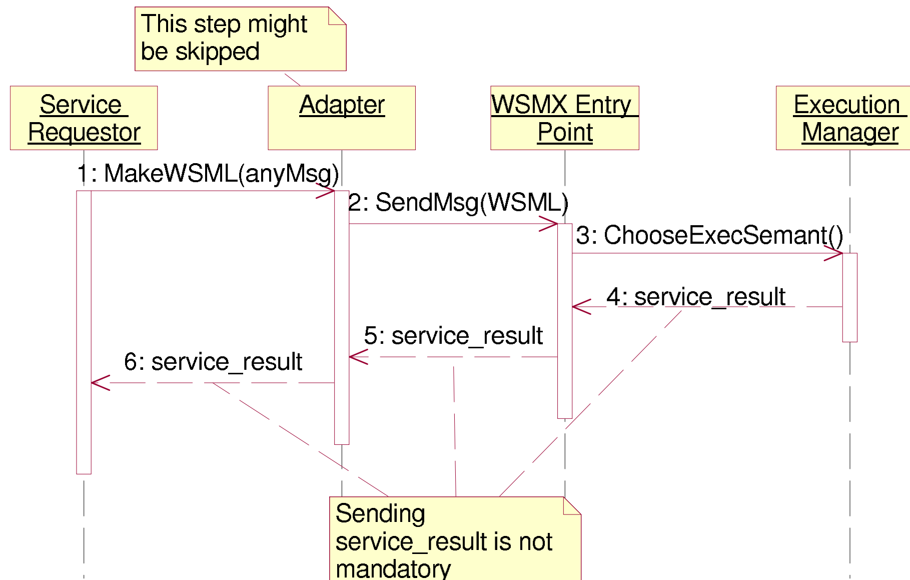

Figure 1: General entry point selection path

of the various component owners. Components behaviour and interfaces should follow the guidelines described in WSMX architecture document [7].

4.1 One-way goal execution

The following entry-point initiates this system behavior.
realizeGoal(Goal, OntologyInstance):Confirmation

The service requester, which expects WSMX to discover and invoke the Web Service without exchanging additional messages can use this entry-point by providing the formal description of Goal (in WSMO terms) and a fragment instance of Ontology. This simplified scenario is based on the assumption that service requester has prior knowledge of all the data required by the Web Service provider. WSMX discovers and executes the service on behalf of requester. If necessary data mediation can be performed. It is not mandatory to send a final confirmation by WSMX since some entities might not have permanent addressing (e.g. the request could be sent from a dynamic IP Internet connection and the user might have already shut down his connection). Usage of this execution semantic is very restricted and in real live situations it is preferable to use it for testing purposes or for fixed services bindings only.

In Figure 2 the definition of the system is given. The behaviour of some components is specified in subsequent diagrams, which is denoted by a blue rectangle underneath the transition (e.g. discovery subnet).

First, a list of *known web services* is created by combining internally known Web Services with external ones in a case the service requester wants to check them too. From this list, one Web Service is picked and an attempt made at matching. If necessary, data mediation is requested, by placing a token in the place *need data mediation*. This mediation may succeed, after which the matching can continue. The mediation may also fail, after which a new Web Service is needed (the chosen Web Service cannot be mediated, and is useless

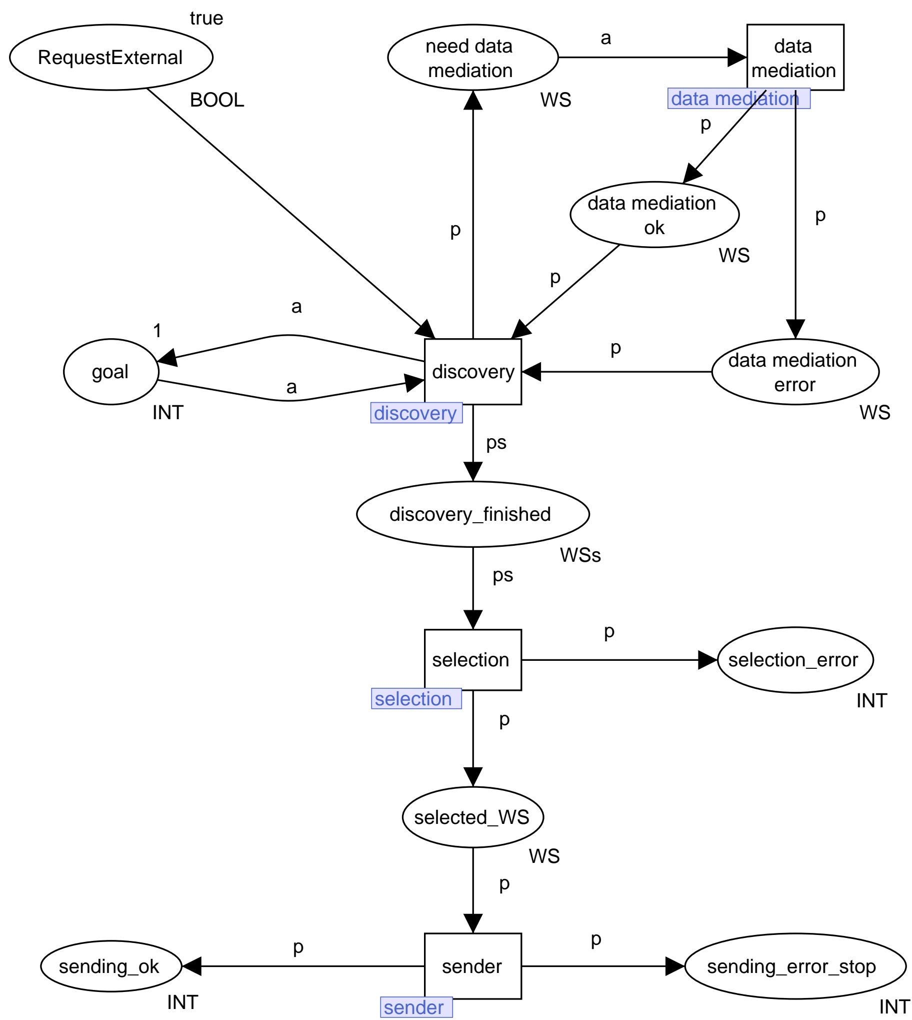

Figure 2: Overview of one-way goal execution

for this goal).

The discovery process continues (with or without data mediation) until a list of Web Services is completed. The list of Web Services is completed when a specific number of Web services is collected or if there are no more Web Services on the *known_WSs* list from discovery subnet 3. The list of found Web Services is returned as a result of the discovery process. The list will be empty if no Web Services are found. This means that all the Web Services have been tried for matching, but none of them succeeded.

After discovery the *selection* component selects the Web service that best fits the user's preferences. Finally, the *sender* component sends a message (invokes) the selected Web service.

The process of discovery 3 includes matching transition. This models the range of possibilities that can occur when deciding whether a web service matches a goal. There are three possibilities here: either there is a matching (denoted by *matching ok*), or there is no matching which means the discovery should retry using another web service (denoted by *matching error*). The third possibility is that data mediation is needed, denoted by placing a token in *need data mediation*, and waiting until this mediation is successfully finished *data mediation ok*.

The matching is (from the viewpoint of the WSMX system) a nondeterministic choice, either a matching is found, or an error occurs or a data mediation is needed; this choice is made not by the WSMX system, but by the *matching* process that infers on user's goal and Web Service semantic description.

This nondeterministic exclusive OR is modelled by having an output place *matching_out*, whose token is consumed by either by the *matching_ok* transition, or by the *matching error* transition, or by the *need data mediation* transition.

The same pattern is repeated for modelling the other components, all of whose outcomes are nondeterministic exclusive ORs (from the viewpoint of WSMX). Both the *data mediation* component and the *selection* component can either fail or succeed, from the viewpoint of WSMX. This is modelled in Figures 4 and 5.

The *sender* component is responsible for sending outgoing messages. Seamless approach to sending messages is provided since either service requestor or service found by WSMX expose their interfaces via Semantic Web Services (initially probably by WSDL). Communication with both of them is carried out in asynchronous manner, i.e. Sender component does not block in awaiting for response. Address of Receiver component and unique ID are included in outgoing message header, thus recipient's response can be send back to Receiver component and be explicitly associated with context of conversation. asynchronous Web services invocations are currently addressed by WS-Addressing initiative[2]. If any of interacting parties communicates synchronously Communication Adapter needs to be utilized as a intermediary layer located outside of WSMX, thus any blocking operations are carried out by the Adapter.

4.2 List of Web Services fulfilling a given Goal

The following entry-point initiates the creation of the list.
receiveGoal(Goal, OntologyInstance, Preferences):WebService[]

A list of Web Services is created for a given Goal and Instance of Ontology. Additionally in Preferences the requester can specify the number of Web Services to be returned. Only Discovery and Data Mediation components carry out their tasks. Each Web Service in the WebServices[] list is already mediated to the

Figure 3: Discovery inside WSMX

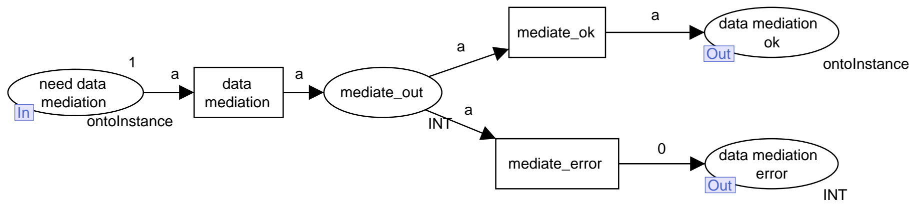

Figure 4: Data Mediation inside WSMX

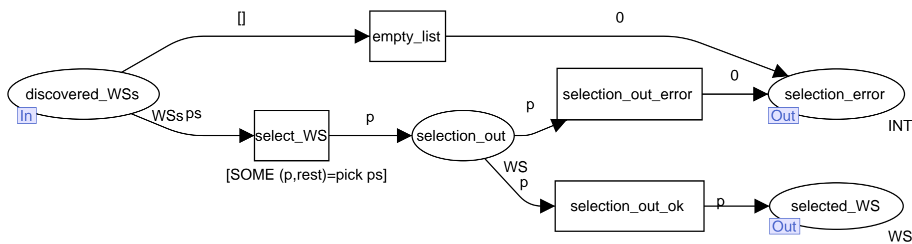

Figure 5: Selection inside WSMX

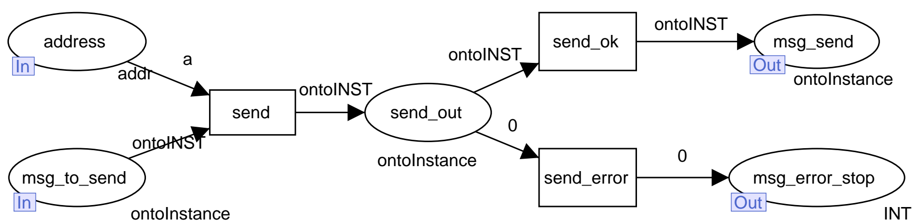

Figure 6: Message sending inside WSMX

requester Ontology if necessary and has its choreography [4]. Since the returned list of Web Services is specified in the same Ontology as the requester's one, the requester can understand them and make a choice which one should be executed. Exchange message patterns are specificated by choreography, thus both parties know what data and in what order is required to carry out the requested service functionality.

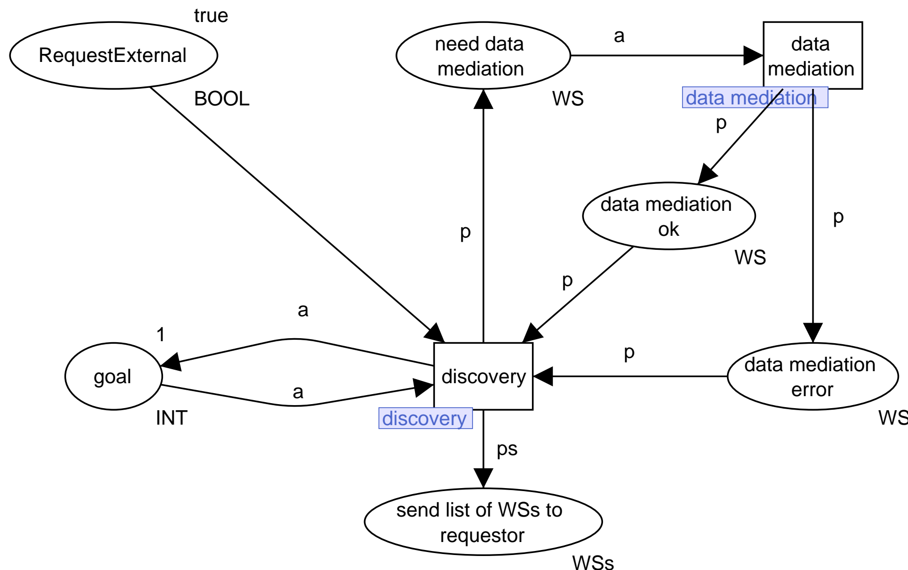

Figure 7: Overview of list of Web Services fulfilling given Goal

This execution semantic is quite relevant when a decision about which Web Service to execute has to be made outside WSMX system. The decision could be taken manually or by some Web Service evaluation program.

The process of generating a list of Web Services fulfilling a given Goal is depicted in Fig. 7. The discovery subnet is running in loop until the desired number of Web Services is collected or until there are no more Web Services to discover. Finally, list of discovered Web Services is sent to the service requestor.

4.3 Web Service execution with choreography

The following entry-point initiates this execution semantic:
receiveMessage(OntologyInstance, WebServiceID, ChoreographyID):ChoreographyID

Once the service requester knows which Web Service he wants to use, back-and-forth conversation has to be carried out with the WSMX system to provide all the necessary data to make the execution of this Web Service feasible. This Execution Semantic involves Process Mediation [3] between interacting parties. By giving fragments of Ontology Instances (e.g. business documents such as Catalogue Items or Purchase Orders in a given ontology) it provides all the data required by the Web Service.

As presented on Fig. 8 conversation between parties takes place according to requestor and Web service choreography. The first time this entry point is invoked, requestor and the given Web service choreography are internally instantiated regardless of their own internal choreographies. This approach facilitates keeping track of ongoing conversation status. They are both referred

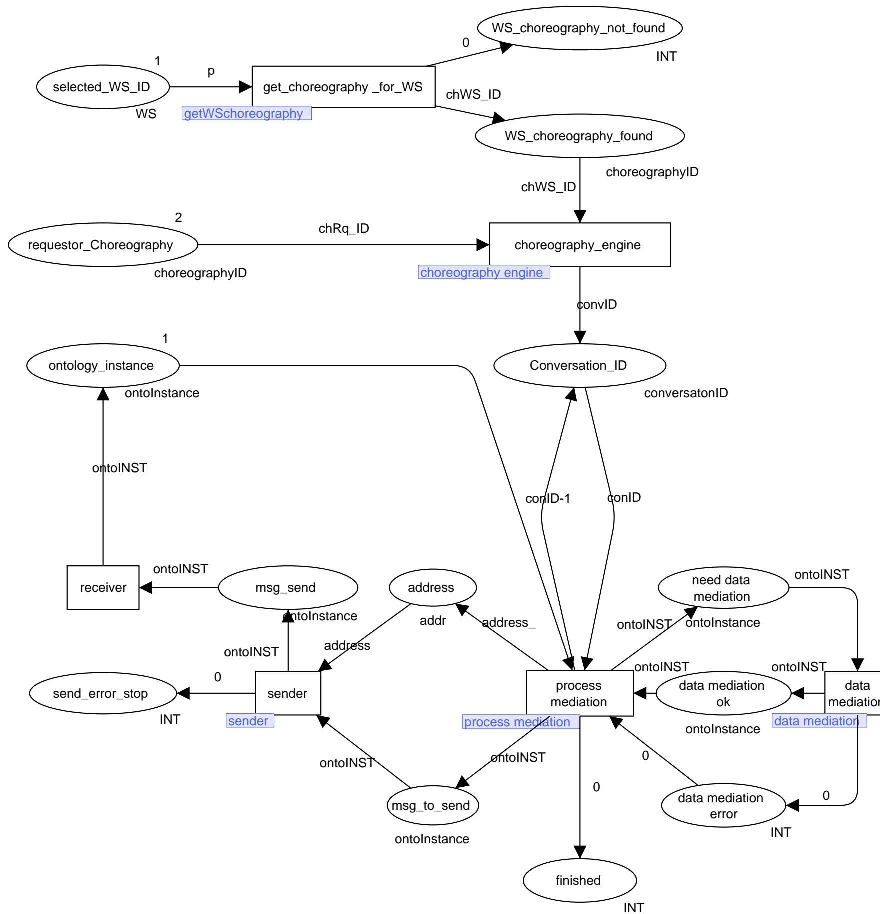

Figure 8: Overview of Web Service execution with choreography

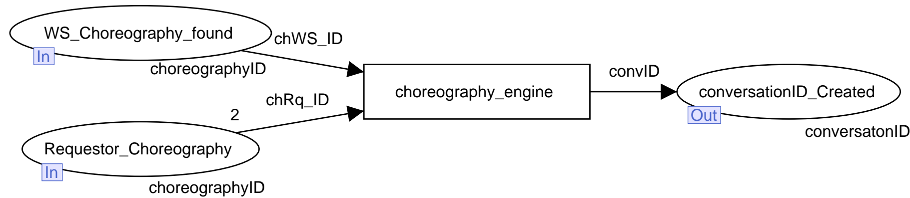

Figure 9: Choreography Engine

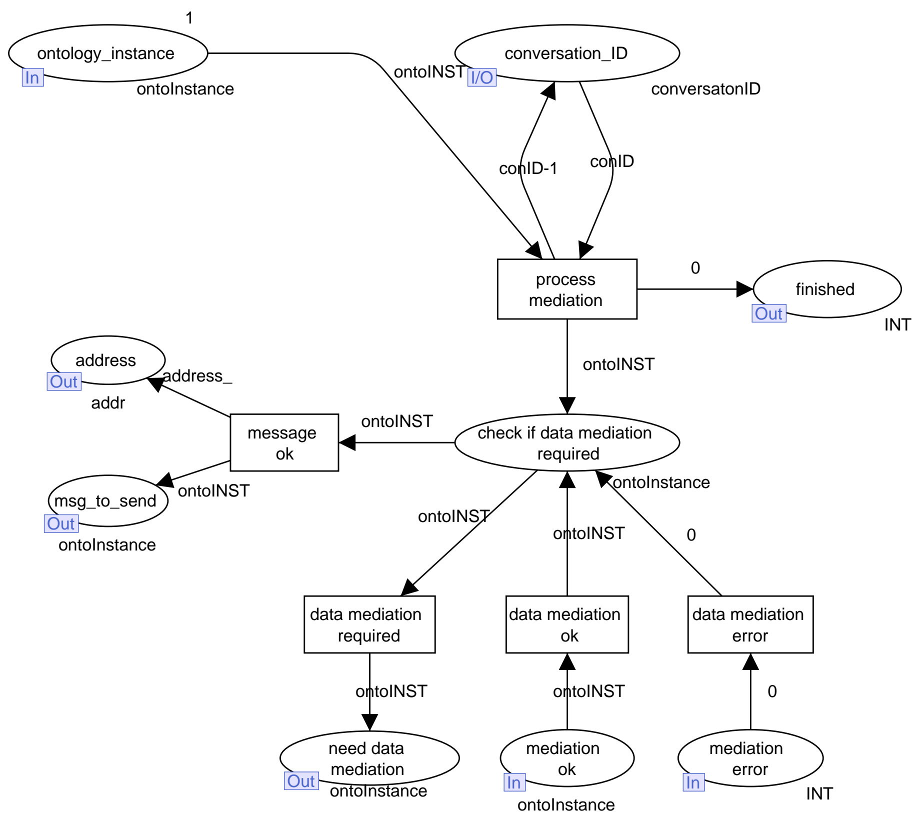

Figure 10: Process Mediation inside WSMX

to their unique identifiers, i.e. *chWS_ID* and *chRq_ID*.

In the next step, as depicted on Fig. 9, a unique ConversationID is assigned to both choreographies, since they belong to the same conversation context and can be treated as a pair. This ConversationID allows one to keep track of ongoing conversation between parties. That is, it keeps a current state of both sides of the choreographies and preserves additional data related to Process Mediation (e.g. messages to be send).

Next, Process Mediation (Fig. 10) is invoked in a loop, thus it goes through all required states for requestor choreography. Process Mediation allows mitigation of differences between choreographies of interacting parties (in this case requestor's and service's). For instance, it can change sequence of messages, generate some dummy messages, preserve some messages to send them later or drop messages. In general, its goal is to enable communication between parties despite of their different expectations of communication patterns (i.e. choreographies). In each invocation of Process Mediation, a ConversationID is used to determine currently processed steps and to update interacting choreographies. Process Mediation can also require some additional data preserved in conversation context pointed by ConversationID, such as messages to be sent in different order or previous actions.

Process Mediation as an input requires ConversationID and Ontology Instance. If necessary the Data Mediation component is requested to mediate Ontology Instance to target ontology (i.e. message recipients ontology). Ready message and recipient address is forwarded to the Sender component. Next, a message is sent asynchronously and in a message header the address of the Receiver component is included and also the unique ID. Message is received by the the Receiver component and then it is again passed on to the Process Mediation together with the conversationID. Process Mediation, then updates the states of the choreographies and the the conversation is carried on.

It should be stressed that the approach presented in this Execution Semantic allows seamless communication with the outside world. One type of Sender and Receiver component can be exploited for communication with both parties. From a point of view of these components, there is no differentiation in interacting with requestor or Web service. It is necessary to distinguish between them in process mediation, where requestor's choreography should have higher priority. However, it is still up to component provider how to exactly model its internal behaviour.

4.4 Store WSMO entity

The following entry-point initiates this execution semantic:
storeEntity(WSMOEntity):Confirmation

The Store Entity entry-point provides an administration interface for the system, enabling it to store any WSMO-related entities (like Web Services, Goals, Ontologies) and making them available for other parties using the WSMX system.

5 Dynamic execution semantics

Since WSMX consists of loosely-coupled components, one could easily imagine other execution semantics that will appear in the future. Thanks to its

architecture it can be easily enhanced with new components. Components can be dynamically plugged in or plugged out to the system. New versions of components can replace outdated ones in this manner. Components can be deployed on remote machines and still be able to subscribe to WSMX events through Tuple Space and to process them. This gives the designer a flexible way to create new execution semantics.

The new executions semantics specifications should not be restricted to one language. They should even not be based on one paradigm (in this deliverable we consider only colour Petri Net-based representation). In order to create interoperable WSMX systems, WSMX should take advantage of efforts like M3PE or WFMC with its XPDL initiative. These efforts are concerned with creating interoperable workflow language that other languages would be able to map to. Their ultimate goal is to be able to execute any workflow specification within one engine.

6 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; by the Vienna city government under the CoOperate programme and by the FIT-IT (Forschung, Innovation, Technologie - Informationstechnologie) under the projects RW^2 and TSC.

The editors would like to thank to all the members of the WSMO, WSML, and WSMX working groups for their advice and input into this document.

References

[1] W.M.P. van der Aalst, K.M. van Hee, and G.J. Houben. Modelling workflow management systems with high-level Petri nets. In G. De Michelis, C. Ellis, and G. Memmi, editors, *Proceedings of the second Workshop on Computer-Supported Cooperative Work, Petri nets and related formalisms*, pages 31–50, 1994.

[2] D. Box, E. Christensen, et al. Web Services Addressing (WS-Addressing). W3C Member Submission, 2004.

[3] E. Cimpian and A. Mocan. Process Mediation in WSMX. WSMX Working Draft v01, 2005.

[4] Emilia Cimpian, Uwe Keller, Michael Stollberg, and Dieter Fensel. Choreography in WSMO. DERI Working Draft v01, 2004.

[5] B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows*. PhD thesis, Queensland University of Technology, Brisbane, Australia, 2002.

[6] A.V. Rantzer, L. Wells, H.M. Lassen, M. Laursen, J.F. Qvortrup, M.S. Stissing, M. Westergaard, S. Christensen, and K. Jensen. Cpn tools for editing, simulating and analysing coloured petri nets. In W.M.P van der Aalst and E. Best, editors, *Applications and Theory of Petri Nets 2003: 24th International Conference, ICATPN 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 450–462. Springer-Verlag, 2003.

[7] M. Zaremba, M. Moran, and T. Haselwanter. WSMX Architecture. WSMO Working Draft v02, 2005.