# D13.2 WSMX Execution Semantics

Eyal Oren  
29th March 2004

Copyright © 2004 DERI ®, All Rights Reserved. DERI liability, trademark, document use, and software licensing rules apply.

## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td>2</td>
<td>Methodology</td>
</tr>
<tr>
<td>2.1</td>
<td>What is execution semantics</td>
</tr>
<tr>
<td>2.2</td>
<td>What is a formal definition</td>
</tr>
<tr>
<td>2.3</td>
<td>Why are we modeling execution semantics</td>
</tr>
<tr>
<td>2.4</td>
<td>How do we model execution semantics</td>
</tr>
<tr>
<td>3</td>
<td>Overview of the Execution Semantics</td>
</tr>
<tr>
<td>3.1</td>
<td>SmallComp and BigComp</td>
</tr>
<tr>
<td>3.2</td>
<td>Feeding WSMX the necessary information</td>
</tr>
<tr>
<td>3.3</td>
<td>Using WSMX for placing purchase orders</td>
</tr>
<tr>
<td>4</td>
<td>Formal model of execution semantics</td>
</tr>
<tr>
<td>4.1</td>
<td>Used modeling technique</td>
</tr>
<tr>
<td>4.2</td>
<td>Used tool</td>
</tr>
<tr>
<td>4.3</td>
<td>Execution semantics of WSMX</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Overview</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Runtime</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Mediation</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Invocation</td>
</tr>
<tr>
<td>5</td>
<td>Acknowledgment</td>
</tr>
</tbody>
</table>

## 1 Introduction

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

WSMX is developed as an example implementation of an execution environment for web services. The goal 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.

In this deliverable we define the execution semantics of WSMX. Accommodating 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 existing literature and choose an appropriate modelling technique. Using this technique we define the execution semantics of WSMX, so that its operational behaviour is formally and unambiguously specified.
2 Methodology

2.1 What is execution semantics

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

2.2 What is a formal definition

A formal definition is a specification in a formal language, which is a language for which a mapping to a mathematical model is defined. In this mathematical model the grounding axioms, the concepts and the relations between them are described. A mathematical model has an interpretation that describes how statements in the model should be interpreted with respect to the real-world domain. A mathematical model is only concerned with abstract concepts, it has no notion of the fact that these abstract concepts represent things in the real world. Statements in a mathematical model are therefore never about the real-world but only about the abstract concepts in the model. The interpretation of this model then relates the abstract concepts and the statements about them to real-world concepts and the statements about these real-world concepts.

In a model certain statements can be deduced from other statements. A model is sound if only true statements can be deduced, i.e. that a deduced statement is really a logical consequence of the existing statements; this means that no false statements can be deduced. A model is complete if every logical consequence can be deduced in the model. So, the completeness property certifies that the model is powerful enough to deduce all that could be deduced logically while the soundness property certifies that no false statements can be deduced.

A formal definition should be sound and complete with respect to the modelled behaviour. If not, it is useless for reasoning because it does not represent the real-world behaviour correctly: it would for instance be possible to deduce statements in the model that are not true in the real-world.

A sound and complete formal definition makes it possible to reason about statements in the language, using proved logical entailments. In that way the behaviour of the system can be checked before (or without) implementing it. If the model is sound and complete it would be possible to prove statements about the system, for instance that there are no unreachable states (i.e., that there are no parts of the system that are not used), or that the system will reach a terminating state eventually (i.e., that there is no livelock).

2.3 Why are we modeling execution semantics

WSMX has two functions in the complete body of WSMO: it serves both as a testbed for WSMO and as an example implementation. This example implementation could for instance be used to demonstrate the viability of WSMO or as a reference for others that want to build their own WSMO execution environment. Contrary to the rest of 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 its work, and to certify that what will be built will satisfy the user’s requirements.

Formal methods can be used during a design process to improve both the results and the process itself. Formal methods are “mathematically-based languages, techniques and tools for specifying and verifying hardware and software systems” (Clarke and Wing 1996). So, the modeling of execution semantics is a formal method for software specification as part of the complete software design process.

Formal methods are used in software specification “to reveal ambiguity, incompleteness and inconsistency” (Wing 1990). In early stages they help to identify design flaws that would otherwise only be discovered (if at all) during testing; repairing these flaws at that stage is usually much more expensive than when they are identified earlier. It cannot be stressed enough that using a formal method does not automatically give you correct programs: “Use of formal methods does not a priori guarantee correctness. However, it can greatly
increase our understanding of a system by revealing inconsistensies, ambiguities and incompletenesses that might otherwise go undetected” (Clarke and Wing 1996).

So, the greatest benefit of using formal methods in the design process comes from the increased understanding of the system and increased agreement between different team members; this is not so much due to the resulting specification but much more to the process of formalizing the individual ideas about the system (Wing 1990). The reason to model the conceptual model and execution semantics prior to the technical software design lies mostly in this benefit: the increased understanding and agreement between team members about the behaviour of the system.

As an added benefit formal methods allow one to (semi) automatically check certain properties of the constructed specification. If the specification is written in a (logical) language that has an inference system, you can derive consequences out of the specification. Taken the specification as a set of facts, you derive new facts (properties) from the application of the inference rules. Using this inference you can prove properties of the specifications that were not explicitly stated, thereby predicting possibly invisible behaviour of the future system. In this way you can test future properties of the system without having to implement it and test it, and reveal properties that would not be discovered during testing. Since testing can only find the presence of errors but never the absence of errors, it cannot be used to proof a property of a system.

To serve as a prescription during the implementation, it is of the utmost importance that the specification is humanly understandable. Otherwise the situation could arise that the specification is perfect, several properties has been checked and verified, but since the developer does not understand the specification correctly, the implementation does not follow the specification and the system will still not behave correctly.

To summarize, formal techniques are used for specification and verification of systems. The reasons are twofold: to enhance the developers’ understanding of the system and to (semi) automatically check properties of the system. To accomodate for the first goal, enhancing the developers’ understanding of the system, the technique must be easily readable by humans, yet unambiguous in its interpretation. This is an important property, on which we will compare several available techniques.

2.4 How do we model execution semantics

Several methods exist to formally model software behaviour. 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, some are logical; some methods have tool support for modeling, for verification, for simulation or for automatic code generation and others don’t. From all these possibilities, We need to choose one method that suits best for describing our software.

We shall now briefly describe some often used methods for specifying software. After that, a justified choice will be made.

Z  todo

CSP  todo

CCS (π-algebra)  todo

Abstract State Machines  todo

Petri nets  todo

To choose one method, we first define our requirements: first of all, the method should be as expressive as needed to define 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 modeled system. Lastly, since some methods are better suited for modeling a certain problem domain than others; the method should be suitable to model our specific problem domain (Kiepuszewski 2002).

Not all these methods support modeling concurrent processes. Although it it’s current state WSMX is a sequential, stand-alone, software system, this will presumably change in the future. When WSMX will support stateful conversations (which is planned for later versions) it will become a distributed system; the several WSMX’s that are conversating together can be seen as (distributed) concurrent processes. Therefore it is wise to use a modeling technique that can handle concurrency well, e.g. CSP, CCS or Petri nets.
3 Overview of the Execution Semantics

In this section we will give an overview of the execution semantics of WSMX. The complete model will be presented in chapter 4, this section serves as an overview and introduction to this model. Let us begin with a small scenario, of the possible future use of WSMX. Please note that WSMX is only works for a specific subset of WSMO, namely WSMO-Lite; future versions will however support WSMO-Standard.

3.1 SmallComp and BigComp

SmallComp is a little company that tries to get into the automobile business; they want to purchase parts from different vendors, and combine them into a complete car. For this business plan to work, they do not only need an assembly line for the parts, but they also need to make sure that all suppliers understand exactly what SmallComp wants from them. One of these suppliers is BigComp, and since BigComp is quite big they prescribe to SmallComp how to place purchase orders with them: they offer a web service that SmallComp can invoke to place their purchase order, and this web service only understands orders with a certain format and a specific terminology. However, SmallComp knows from experience that they use different terminology than BigComp; for instance, the thing BigComp calls an *engine*, SmallComp is used to call a *powersource*.

SmallComp decides to use WSMX to talk to BigComp in an easy way. Luckily for SmallComp, BigComp has written a WSMO specification describing their web service, the capability that this web service offers, and the ontology that this web service uses. The capability of this web service is to accept purchase orders and promising a delivery date. The ontology consists of concepts like *Purchase Order*, *Message*, *Car*, *Engine*, *Wheel*. So the ontology is not only a domain ontology (describing engines and wheels), it also describes the communication (purchase order, message).

3.2 Feeding WSMX the necessary information

This WSMO specification that BigComp wrote is fed to the WSMX of SmallComp. This specification is checked (both for syntactical correctness and to see wether all referenced elements are known) and stored persistently somewhere in the system. From now on, the WSMX at SmallComp ‘knows’ about BigComp and what kind of service they offer. Now somebody at SmallComp feeds the WSMX an *integration type*, describing their goal, which is placing a purchase order and being promised of delivery, and the ontology they use (*Purchase Order*, *Message*, *Automobile*, *PowerSource*, *Wheel*). This specification is also checked and stored in the system. From now on WSMX ‘knows’ about this integration SmallComp set up with BigComp.

Now, only one thing must be done before SmallComp can start sending purchase orders. Because the ontologies that SmallComp and BigComp use are not the same, mediation rules must be defined to map the concepts from one ontology to the other. Somebody at SmallComp looks at their ontology, takes a good look at the ontology from BigComp, calls somebody in BigComp to ask what they mean by *Car* etc., and writes the rules to map one or more concepts from one ontology to the other. These rules are simple logical expressions and to accomodate the user a tool is available to map concepts and attributes graphically; writing the mapping rules by hand is only necessary for a small part of the rules. After feeding these rules into WSMX the sending of purchase orders is possible.

3.3 Using WSMX for placing purchase orders

In day-to-day business, some back-end application (for instance some planning system) can call a web service on WSMX and invoke the operation *execute integration instance*. This back-end application specifies which integration type it wants to use (there may be many integrations specified in this system, not only to BigComp, and not only about purchase orders), and supplies all the values for the specific purchase order it wants to place with BigComp.

Since an integration type is linked to a goal, and a goal matches exactly one capability (in WSMO-Lite), the execution environment selects all possible web services that provide the capability the integration type is asking for, and selects one. In the current version of WSMX this selection is done randomly, in the future selection criteria can be specified on non-functional properties (like price, reliability, etc.).

After selecting one web service, the execution environment can look up which ontology this web service uses. When this ontology differs from the one SmallComp is using (which it would normally do) the execution
environment needs to mediate between those to. For this mediation task a logic reasoner will be used. Since this reasoner uses a different format than WSMX, a (syntactical) translation must be performed. After that, the instance is fed to the reasoner, specifying the result should be in the ontology of BigComp. If the reasoner can perform a mediation, his answer will be translated again into the WSMX internal format. If the reasoner cannot perform a mediation, the back-end application will be notified of an error: either it didn’t supply the right or enough information for this integration type, or there are not enough mediation rules specified for these two ontologies.

Because both the message SmallComp wants to send and the message BigComp's web service expects is specified in their ontologies, mediation between the first and the second has taken place. So, if mediation has taken place successfully BigComp’s web service will be invoked by WSMX and the message (mediated to a format BigComp can understand) will be sent in this invocation. If something goes wrong in this invocation (BigComp's web service does not respond, or it responds with an error message) the back-end application will be notified.

Every state change of this specific integration instance is recorded persistently somewhere in the system, so to keep track of the execution history of every single integration instance.

4 Formal model of execution semantics

The scenario as described in section 3 describes one way to interact with WSMX, and the behaviour of WSMX during this interaction. The specification of the complete execution semantics takes into account not only this single scenario, but all possible ways to interact with WSMX, and the behaviour of WSMX during on possible events during such an interaction.

In this section we present the modeled definition of the execution semantics of WSMX. Please note that this is not yet the definite version. Several things are missing in the model; this will be added in the future. Both the feeding of integration types to WSMX and the feeding of known web services to WSMX will be added in future versions. Secondly, more errors can occur than is modeled right now, which will be corrected in future versions. Also note that not all transition definitions are semantically correct now; they form a correct Petri net and describe some behaviour, but this is not yet the desired behaviour. This only shows that modeling execution semantics can indeed be done incrementally, as it should be.

4.1 Used modeling technique

We will describe the execution semantics using hierarchical, coloured Petri nets. 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 said otherwise, a transition now becomes a relation between the value of the input tokens and the value of the output tokens (Aalst et al. 1994). In addition, a transition can state conditions over its input tokens, that must be satisfied before the transition can become enabled.

We are using hierarchical Petri nets, which means a transition can be decomposed into a so-called subnet. The subnet must have as many input and output places as the decomposed transition. Using this extension it is possible to break down large models into smaller pieces, and to model incrementally. A transition can first be modeled as elementary, while later on decompositing it into more detail. It is important to note that adding the concepts of both colour and hierarchy to Petri nets does not improve their expressivity, but only makes them more readable.

The tool offers the possibility of defining timed Petri nets, which is an extension introducing a global time, and a timestamp on every token, providing the possibility to describe time duration of transitions. We will not make use of this possibility.

4.2 Used tool

The tool we are using, CPNTools (Rantzer et al. 2003) 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 modeler’s understanding of the system. When a modeler is not completely well-known with Petri nets, it is quite easy to construct a Petri net that does not follow the modeler’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 modeled system in detail.

4.3 Execution semantics of WSMX

4.3.1 Overview

In figure 1 the high level definition of the system is given. If an instance invocation is done, given a certain instance type and some values, an integration instance is created. This integration instance is executed, and if no errors occur, this instance execution is completed. The empty web service that is shown is a modeling artifact, it’s use will be explained later. As this is a coloured Petri net, all places are of a certain type, which means that in only tokens of that specific type can belong to that place. The relation between the values of the input places and the values of the output places is specified on the (input and output) arcs of each transition. For instance, in the transition create_instance the input integration_type, value and webservice are combined into one integration instance.

The tool we are using uses a functional language (CPN ML, an extension of Standard ML) for declarations and inscriptions. Declarations can be written by the modeler to define types, functions and variables (independent of a specific Petri net). Inscriptions can be defined for places, arcs and transitions. Place inscriptions define the type of the place, for instance the places integration_instance, error and executed integration are all of type integrationInstance; they can also be used to describe initial markings, that is the tokens that are available in a place when starting a simulation. Arc inscriptions define conditions on the input and output of transitions. Transition inscriptions can be used to define guards, time delays, and (more complex) relations between the input and output of the transition. Transition inscriptions are not shown in this figure.

4.3.2 Runtime

As can be seen in the figure, the execution of an instance is decomposed in another figure, called ‘runtime’. This decomposition is shown in figure 2, that has exactly the same input and output places as the decomposed
transition.

For the integration instance to be executed, the first thing that has to be determined is the web service that has to be invoked for this instance. Since WSMX is concerned with the subset of WSMO in which goal consist of exactly one capability, the selecton of a web service consists of finding a web service that provides a capability that is equal to the goal of the integration instance. In case such a web service can be found, mediation has to take place, to translate between the source ontology (used by the organization that hosts this WSMX) and the target ontology (used by the web service that will be called).

If that has been done succesfully, the web service can be invoked, passing the mediated message. We are again dealing with a subset of WSMO, in which no conversations exist between web services, but only simple requests and responses. Therefore, there is no need to wait (or otherwise exspect) for an answer after invocation, and after this has been recorded in the history, the execution of this instance is done.

4.3.3 Mediation

The actual mediation that in figure 2 is decomposed into a subnet. This subnet is shown in figure 3.

The mediation takes place mostly outside the scope of the execution semantics. We are using a logical reasoner to perform this mediation and we will not model the behaviour of this reasoner. We feed this reasoner with mediation rules and a set of facts, and ask him for the resulting facts. The execution semantics of these logic programs (that is what this mediation basically is) lies outside the scope of this document, mostly because the formal semantics of these logic programs has been described elsewhere. The question is of course of the specific reasoner adheres to this formal semantics, but that is a different exercise.

What basically needs to be done by WSMX before invoking the reasoner, is translating the instance data from its internal format to some format the reasoner can understand. Next, it has to determine the ontology that is used by the web service selected for invocation. In case the reasoner can make a succesfull mediation, this answer needs to be translated back into the internal format of WSMX. It is very well possible that the reasoner cannot find a mediation between these two ontologies, for instance, if not enough (or none at all) mediation rules have been specified, or not enough data has been specified to perform the mediation.
Figure 3: Execution semantics of WSMX: Mediation

Figure 4: Execution semantics of WSMX: Invocation

4.3.4 Invocation

The mechanism for invocation in figure 2 is also decomposed into a subnet, which is shown in figure 4.

For the selected web service to be invoked, the mediated data has first to be transformed in some format that can be sent over the Internet (e.g. some SOAP message). Then the selected web service has to be invoked (using the grounding that was specified for this web service), after which three things can happen: first, the web service might not respond at all, which we can model using a time out error. Secondly, the web service can respond using some (in WSMO) predefined error code, indicating that for some reason it is not able to process this purchase order. If neither of those occurs, the invocation has been successful.
References


URL http://wiki.daimi.au.dk/cpntools/cpntools.wiki


5 Acknowledgment

The work is funded by the European Commission under the projects DIP, Knowledge Web, Ontoweb, SEKT, SWWS, Esperonto and h-TechSight; by Science Foundation Ireland under the DERI-Lion project; and by the Vienna city government under the CoOperate program.

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