WSMO Deliverable

# D13.2 v0.1
# WSMX Execution Semantics

WSMO Working Draft – 31st May 2004

**Authors:**
E. Oren

**Editors:**
E. Oren

**This version:**
http://www.wsmo.org/d13/d13.0/v0.1/20040531
**Latest version:**
http://www.wsmo.org/d13/d13.0/v0.1/
**Previous version:**
http://www.wsmo.org/d13/d13.0/v0.1/20040427

# 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 mod-

elled 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 live-lock).

## 2.3 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 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"* [2]. So, the modelling 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"* [8]. 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 inconsistencies, ambiguities and incompletenesses that might otherwise go undetected"* [2].

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 formalising the individual ideas about the system [8]. 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 summarise, 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 accommodate 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 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 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 to model our specific problem domain [4].

For choosing a method, two things are therefore most important: to be able to model the problem domain, and to do so as understandable as possible. 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 conversing together can be seen as (distributed) concurrent processes. Therefore it is wise to use a modelling technique that can handle concurrency well, for which we have chosen 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 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 works with a specific conceptual model, WSMX-O; future versions will however support WSMO-Standard.

Secondly, 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', not modelling the 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 of the various component owners.

As a guideline to component owners a description of the preferred behaviour of the components is included. Using this guideline the complete execution semantics of WSMX is specified; it is however up to the component owners to provide an execution semantics that conforms to this guidelines.

For this specification we use the functional language Standard ML [3]. Standard ML is a formalised [5] programming language, which means that the semantics of the language is specified independent of some implementation (of a compiler, interpreter or virtual machine). For the specification of (preferred) behaviour of some system, it is important to use a formalised language, as explained above.

Pleas note that specifying the execution semantics is done as part the design process, mainly to ensure a common understanding of the system between developers. While it is possible to model this software system using Petri-nets (some Petri-nets variants are Turing complete, c.f. [6]), this is not always the most readable and user-understandable way. Since every problem domain should use the method best suited to model its execution semantics, we have chosen to use a functional language for the detail required in the component specification.

Summarising, we have chosen to model the overview execution semantics using Petri-nets, modelling the separate components as 'black-boxes'. The component owners should define the execution semantics of these components. To help those component owners we have specified a preferred behaviour of these components, using Standard ML.

## 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 *power source*.

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 whether 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 a *goal*, describing their goal, which is placing a purchase order and being promised of delivery, some *preferences* they have for a service to use, and the ontology they use (*Purchase Order, Message, Automobile, Power Source, Wheel*). This specification is also checked and stored in the system. From now on WSMX 'knows' about what SmallComp would like to achieve and what BigComp offers.

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 accommodate 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. These rules are given to a *mediation* component, that can be used by WSMX to mediate between these two ontologies.

## 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 to execute a goal. This back-end application specifies the *goal* it wants to achieve (there may be many goals 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.

The execution environment asks the *matching* component to find all the web services that provide the capability that satisfies this goal. It is up to the specification of this *matching* component to decide how to perform this goal-capability matching, in the current specification the *matching* component just selects all the capabilities that are syntactically identical to the goal (not taking logical reasoning into account). This component could be rewritten in the future. If the *matching* component can't find any service that could satisfy the goal, an error is returned.

The *matching* component tries to find matching web services one-by-one. For web services that use a different ontology than the ontology used by the goal, mediation is needed, translating the goal to the ontology of the web service. The *matching* component can ask the *mediation* component to perform this mediation. If the mediation was successful the *matching* component can continue trying to find a match; otherwise, the web service can be disregarded, since no mediation is possible between the goal and this web service. Again, the internal working of this mediation component is not relevant here; that is up to the specification of this component. The only thing relevant for WSMX is that this component mediates (if possible) between the two ontologies.

Then the execution environment asks the *selection* component to select from this list of web services the one that matches best the preferences specified in

the goal. Again, it is up to the specification of this component how to perform this selection; in the current specification the component just returns the first webservice of the list.

Finally, the execution environment asks the *invocation* component to invoke the selected web service, using either the original goal or the mediated goal. If something goes wrong in this invocation (BigComp's web service does not respond, or it responds with an error message) the *invocation* component retries to invoke the web service a number of times. If the web service invocation keeps failing, the execution environment is informed that invocation did not succeed. If all goes well, BigComp received the message.

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

# 4 Formal model of execution semantics

The scenario as described in 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 modelled definition of the execution semantics of WSMX.

## 4.1 Modelling technique

We will describe the execution semantics using classical Petri nets. The used tool, CPNTools [7], offers the possibility to model so-called high-level Petri-nets [1], extending classical Petri nets with hierarchy, colour and time.

Hierarchy adds the possibility to decompose a transition into so-called sub-nets, which allows 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 provides the possibility 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 said otherwise, 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 net, but does make them more concise and readable.

The tool uses a functional language, Standard ML, to describe arc inscriptions and transition guards. Both arc inscriptions and transition guards put conditions on the 'firing' of a transition; only the tokens that satisfy the conditions are considered when determining whether a transition is enabled.

Using Standard ML (and defining the inscriptions in general) tends to clutter the model, making it a bit difficult to read. This could however not be avoided, since the idea is not only to make a readable specification but also to precisely define the execution semantics.

## 4.2 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 well-known 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].

## 4.3 Execution semantics of WSMX

In figure 1 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 (this applies for instance to the *matching* component).

First of all a list of *known web services* exists, which are the web services that WSMX knows about, denoted by the place *known ws*. From this list, one web service is picked and matching is tried. If necessary, mediation is asked for, by placing a token in the place *need mediation*. This mediation can succeed, after which the matching can continue. The mediation can also fail, after which a new web service is needed (the chosen web service cannot be mediated into, and is useless for this goal).

The matching process continues until a useful web service is found (with or without mediation). If no web service is found, the *no more ws* transition is fired, resulting in a *matching error*. This means that all the web services have been tried for the matchmaking, but none resulted in a match.

After matchmaking, the *selection* component selects the web service that best fits the preferences specified in the goal[2]. Finally, the *invocation* component invokes the selected web service.

The process of matchmaking is specified in figure 2. This models that when deciding whether a web service matches a goal, three situations can occur: either there is a match (denoted by *matching ok*), or there is no match which means the matchmaking should retry using another web service (denoted by *matching error, new ws needed*. The third possibility is that mediation is needed, denoted by placing a token in *need mediation*, and waiting until this mediation is successfully finished *mediation ok*.

The matching is (from the viewpoint of the WSMX system) a nondeterministic choice, either a match is found, or an error occurs or a mediation is needed;

[1] the model is available on `wsmx_single_wish.cpn`; the tool itself can be downloaded from `http://wiki.daimi.au.dk/cpntools/`

[2] please note, that in the current version of this specification a successful matchmaking results in exactly one web service; this is however not desired, the matchmaking should return a list of matching web services, after which the selection component selects the most preferable one.

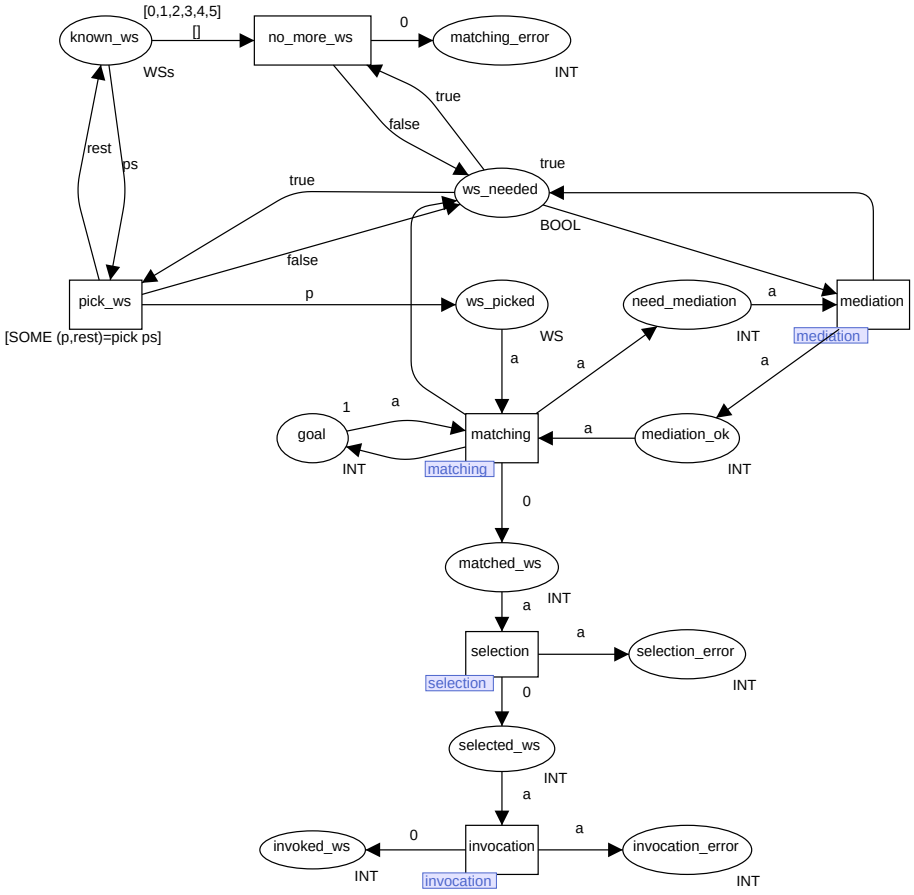


Figure 1: Execution Semantics of WSMX

this choice is made not by the WSMX system, but by the *matching* component. The behaviour of this component is (partly) specified in section 4.4.

This nondeterministic exclusive or is modelled by having an output place *match_out*, whose token is consumed by either by the *match_ok* transition, or by the *matching_error* transition, or by the *need_mediation* transition.

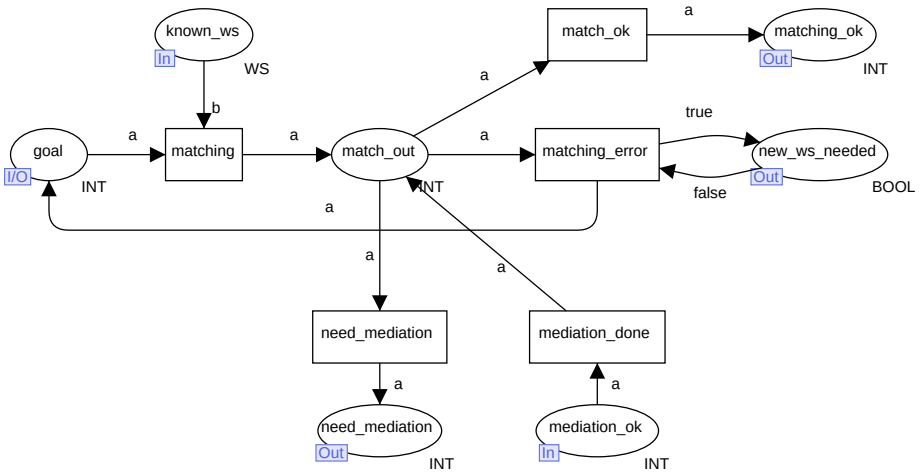


Figure 2: Matchmaking inside WSMX

The same pattern repeats for modelling the other components, all of whose outcomes are nondeterministic exclusive or's (from the viewpoint of WSMX). Both the *mediation* component and the *selection* component can either fail or succeed, from the viewpoint of WSMX. This is modelled in figures 3 and 4.

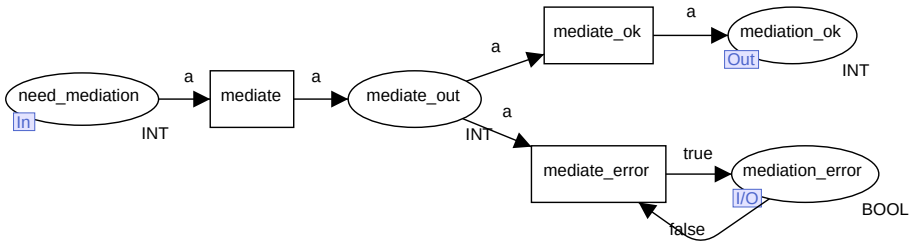


Figure 3: Mediation inside WSMX

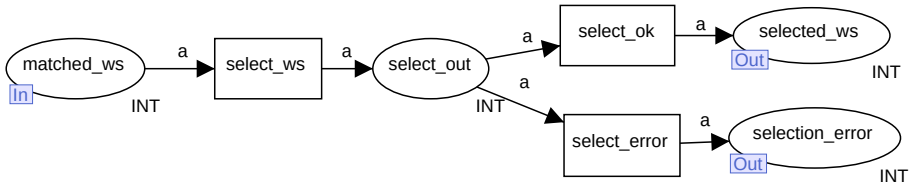


Figure 4: Selection inside WSMX

The *invocation* component is modelled slightly different, since this component to retry invocation a number of times, in case of network time-outs or other temporary errors. This component is shown in figure 5; each time an

error occurs, a counter is incremented. If the counter does not exceed a certain threshold, the invocation is retried; otherwise, the invocation fails.

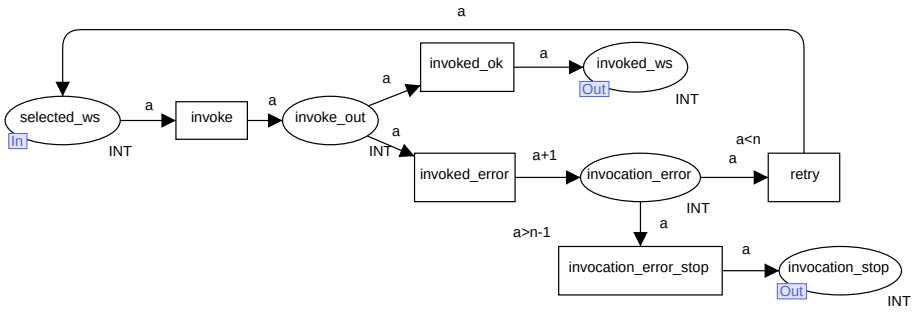


Figure 5: Invocation inside WSMX

## 4.4 Description of preferred component behaviour

In this section the preferred behaviour of the separate components is given, using Standard ML; below the code for this is shown[3].

The main point is that every component is described by a function, that defines the relation between its input and its output. First some data types are defined and some helper functions (that shorten the functional description of the components). Also some test data is defined, useful when running (simulating) this definition.

Next the functions are given describing the preferred component behaviour. The functions have the same input as specified in Petri net model, and can all throw an *exception* (as specified in the Petri net model). However, where the Petri net model sees the functionality of the components as nondeterministic, the functions here specify exactly how the output is computed given the input, and when an exception is thrown.

The *match* function describes how the mapping component works: it filters the list of all known web services, and keeps the web services whose postconditions and effects (actually, the postconditions and effects of the capabilities of these web services) are the same as those specified in goal. In later versions of WSMX this specification should be adjusted, to not only do string matching, but real logical reasoning (including subsumption reasoning).

The *select* function describes how the selection component works: it simply takes the first web service of the list of possible web services it receives. Again, later this specification should be adjusted, to really use the preferences specified in the goal.

The *mediate* function describes the mediation component; however, this component is also in this model a black-box, which is modelled by using an external mediator. The *invoke* function finally, describes how the invoker makes a web service call, using the input specified in the goal, and the interface specified in the web service.

The complete definition can be loaded into a Standard ML compiler, and executed. The behaviour of WSMX (as specified in the Petri net model) can be simulated by executing sequentially the *match, select, mediate* and *invoke* functions.

[3]for an introduction in Standard ML see `http://www.smlnj.org/index.html`.

```
(* exception definition *)
exception error of string;

(* concept definitions, taken from WSMX-O conceptual model
 i didn't (yet) use all of the conceptual model *)
type goal = {postcondition:string, effect:string};
type capability = {precondition: string, assumption: string,
 postcondition:string, effect:string};
type wish = {goal:goal, preference:string, input:string};
type error = {message:string, function:string}

type businesspartner = {name:string};
type interface = {url:string};
type oomediator = {service:string};
type webservice = {capability:capability, interface:interface,
 businesspartner:businesspartner, usedmediator:oomediator};
type message = {sender:businesspartner, receiver:businesspartner,
 content:string};
(* end of concept definitions *)

(* some test data to see this stuff running
 giving an idea of whether the definitions are right *)
val buyBookGoal:goal = {postcondition = "haveBook",
 effect="bookTransported"};
val buyBookWish:wish = {goal = buyBookGoal, preference = "cheap",
 input="harry potter"};
val emptymediator : oomediator = {service = ""};
val amazonCap : capability = {precondition = "money > 0",
 assumption = "stock > 0", postcondition = "haveBook",
 effect="bookTransported"};
val irishrailCap:capability = {precondition = "", assumption = "",
 postcondition = "knowRailTimeTable", effect = ""};

val amazon : businesspartner = {name = "Amazon"};
val amazonInterface : interface =
 {url = "http://www.amazon.com/service.wsdl"};
val amazonService:webservice= {capability = amazonCap,
 interface = amazonInterface, businesspartner = amazon,
 usedmediator=emptymediator};
val proxis : businesspartner = {name = "Proxis"};
val proxisInterface:interface =
 {url = "http://www.proxis.be/service.wsdl"};
val proxisService:webservice = {capability = amazonCap,
 interface = proxisInterface, businesspartner = proxis,
 usedmediator = emptymediator};
val irishrail:businesspartner = {name = "Irish Rail"};
val irishrailInterface:interface =
 {url = "http://www.rail.ie/service.wsdl"};
val irishrailService:webservice = {capability = irishrailCap,
 interface = irishrailInterface, businesspartner = irishrail,
 sedmediator=emptymediator};
val knownWS = [amazonService, proxisService, irishrailService];
(* end of test data *)

(* some helper functions *)
(* filters any list on a condition p, that takes a parameter x
 those elements of the list, for who p(x) is true, go in the returned list
 used for filtering the web services that have certain capabilities *)
fun myfilter (p,x,[]) = []
```

```
  | myfilter (p,x, h::t) =
    if p(h,x) then h :: myfilter (p,x, t) else myfilter (p,x,t);

(* checks whether this webservice has a capability that
 exactly matches the goal (this should of course be refined later on *)
fun checkWSmatchesGoal (ws:webservice, wish:wish) =
 (#postcondition (#capability ws) = #postcondition (#goal wish))
 andalso (#effect (#capability ws) = #effect (#goal wish));

(* placeholder for a call to an external mediator,
 does nothing more than doubling the input, e.g. a -> a-a *)
fun externalMediator (input, ws:webservice) = input^"-"^input;

(* placeholder for a web service call,
 does nothing more than returning the sentence 'made call' *)
fun makecall (interface:interface,message) = ("made WS call,
 saying " ^ message ^ " to " ^ #url interface);
(* end of helper functions *)

(* the real functions*)
(* selects all the web services that have a matching capability *)
fun match (wish:wish, knownWS:webservice list) =
    if knownWS = [] then raise error("no known services")
    else
    let
        val matched = myfilter(checkWSmatchesGoal,wish,knownWS)
    in
        if  matched = [] then raise
                error("no matching service") else (wish,matched)
    end;

(* selects the first of the possible web services,
 will later be done on some nonfunctional properties
 of the web service, and the specified preference *)
fun select (wish, hd::tl) = (wish,hd);

(* transforms the input of the wish using an externalMediator *)
fun mediate (oldwish:wish, ws:webservice) =
 let
     val newInput = externalMediator(#input oldwish,ws)
     val newWish:wish = {goal=(#goal oldwish),
                         preference=(#preference oldwish),
                         input=newInput}
 in
     (newWish,ws)
 end;

(* invokes the web service using the transformed message *)
fun invoke (wish:wish,ws:webservice) =
 makecall(#interface ws, #input wish);

(* this is the main function, showing how a wish
 gets transformed into a web service invocation *)

(* this variables show you what happens in the
 separate execution steps, you can check
 the values to see what happened *)
val matched = match(buyBookWish, knownWS);
val selected = select(matched);
```

```
val mediated = mediate(selected);
val invoked = invoke(mediated);
```

## 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] E.M. Clarke and J.M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[3] R. W. Harper, D. B. MacQueen, and R. Milner. Standard ML. Report ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, Edinburgh, UK, 1986.

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

[5] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1989.

[6] J.L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, September 1977.

[7] 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.

[8] J. M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–26, September 1990.


## 5 Acknowledgement


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 editor likes to thank to all the members of the WSMO working group for their advice and input into this document.