WSMX Execution Semantics: Executable Software Specification

Eyal Oren*

Digital Enterprise Research Institute (DERI), Galway, Ireland
eyal.oren@deri.org

Abstract WSMX is an execution environment for dynamic discovery, selection, mediation and invocation of web services. WSMX builds on WSMO, a conceptual framework for semantically describing web services, goals, ontologies and mediators. The design process of WSMX included formally specifying the operational behaviour of the system. In general, the reasons to formally model system behaviour in a design process are: enlarging developers' understanding of the system, proving several properties of the (model of the) future system and enabling model-driven execution of components. We present the execution semantics of WSMX and describe whether our approach addressed these requirements.

1 Introduction

The Web Services Execution Environment[1] (WSMX) is an execution environment for dynamic discovery, selection, mediation and invocation of semantic web services. WSMX builds on the Web Services Modelling Ontology[2] (WSMO) that describes various aspects related to semantic web services.

WSMO is based on four concepts: web services, ontologies, goals and mediators. *Web services* are units of functionality; every web service has exactly one capability, that describes logically what this web service can offer. Every web service has a number of interfaces, which specify how to communicate with it. *Goals* describe some state that a user may want to achieve. *Ontologies* are the formal specification of the knowledge domain used by both the web service to express its capability, and by the goal to express the desired world state. *Mediators* are used to solve different interoperability problems, like differences in ontologies used by a web service and a goal.

WSMX is developed as an reference implementation of an execution environment for web services. WSMX manages a repository of web services, ontologies and mediators. WSMX can achieve a user's goal by dynamically selecting a matching web service, mediating the data that needs to be communicated to this service and invoking it.

* This material is based upon works supported by the Science Foundation Ireland under Grant No. 02/CE1/I131.

[1] see http://www.wsmx.org

[2] see http://www.wsmo.org

In the design process of WSMX several steps were taken, including designing an architecture, describing a conceptual model and specifying the execution semantics. Execution semantics is the formal specification of the operational behaviour of a system. Such formal specifications can be used for a number of reasons during software development. In the context of WSMX we are interested in modelling the execution semantics of WSMX in such a way, that (i) it helps developers understand the system, (ii) it allows for deducing certain properties of the system and (iii) it enables model-driven execution of components.

Most work in the field of formal software specification and verification is related to (i) and (ii) but in general not concerned with executing specifications. In the field of business process management and web service composition on the other hand, (iii) is investigated (how to enable model-driven execution of components) but (i) and (ii) are usually neglected. We will explain why we believe combining these approaches and using a single formal specification for these three objectives is useful.

The structure of the paper is as follows: first we describe what we mean by execution semantics and outline the advantages of specifying it during software development; then we present the execution semantics of WSMX and describe whether the approach taken was an adequate response to the above-mentioned requirements; we conclude with a brief description of future steps we want to take to overcome the encountered problems.

2 What are execution semantics

In developing software one can distinguish four phases: requirements analysis, software design, implementation and testing. A software design is an answer to the requirements analysis and a guideline for the implementation. During a design process formal methods can be used to improve the resulting design and the process itself. A formal method is a mathematically-based language, technique or tool for specifying and verifying hardware and software systems [4]. Modelling execution semantics is an example of such a formal method.

Execution semantics is the formal definition of the operational behaviour of a system. It describes how the system behaves, what the semantics are of the execution of the system. The formal definition is only concerned with abstract concepts. Statements are not about the real world but only about the abstract concepts in the model. In a model certain statements can be deduced from other statements. A model is sound if only true statements can be deduced, a model is complete if every logical consequence can be deduced in the model. A formal definition should be sound and complete with respect to the modelled behaviour.

3 Why model execution semantics

Formal methods can be used in software specification to reveal ambiguity, incompleteness and inconsistency [13]. In early stages of the development 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. Using a formal method does not automatically result in 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"* [4].

The first 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. An important reason for modelling the conceptual model and execution semantics prior to the technical software design lies in this benefit: the increased understanding and agreement between team members about the behaviour of the system.

To serve as a prescription during the implementation, it is important that the specification is easily human readable. The people that will implement the designed system have to understand the specification in order to follow it. When the implementation differs from the specification the model is not longer related to the real system – all one can prove in the model is then useless in reality.

A second benefit is that most formal methods allow for automatically checking certain properties of the constructed specification. If the specification is written in a language that has an inference system one can derive consequences from the specification. This inference proves properties of the specifications that were not explicitly stated, for instance that no unreachable states exist or that the system will eventually reach a terminating state. In this way one can discover future properties of the system before or without implementing it and reveal properties that might not be discovered during testing.

In a component-based system a third benefit arises. In these systems, different components work together to achieve the functionality of the system, usually managed by some central component. The primary task of this management component is to control and coordinate the execution of the components; the control flow between the components is usually hard-coded into this component. This is not a very flexible situation, for every time the inter-operation of the components changes this management component has to be reprogrammed.

This inflexibility of component-based systems could be overcome by making use of the formal specification of the system: formally specifying the interplay between the different components and using a process enactment service to execute this specification (runtime execution of the modelled system by an engine). The specification would serve as process model stating which component should do what and when; a business process management engine could be used to execute this specification.

To summarise, formal techniques are used for specification and verification of systems. The reasons are threefold: to enhance the developers' understanding of the system, to automatically check properties of the system and to isolate and automate the control-flow between components. To accommodate for the

first goal the technique must be easily readable by humans yet unambiguous in its interpretation. To accommodate for the second goal the technique must be formal and have a proof system in which properties of a model can be deduced. To accommodate for the third goal the technique must have an operationalisation, an execution procedure for a model.

4 How to model execution semantics

We are interested in automating the control-flow between components, as stated in section 3. We will therefore not model the execution semantics of the complete system but that of the central management component. Other components are treated as black-boxes: we know their functionality but not how they operate to achieve this functionality. Therefore we can not describe or predict under which conditions these components will succeed or fail, that is a non-deterministic choice from our viewpoint. What we do model however, is how the management component reacts to the outcome of this non-deterministic choice. We will limit the model to the execution semantics of achieving a goal using WSMX; the behaviour when adding or removing descriptions from the repository is not modelled.

We will describe the execution semantics using Petri nets [10]. Petri nets are a formalism for modelling dynamic systems. They are graphical and therefore (supposedly) easily understood and communicated. They are mathematically formalised and well analysable. Classical Petri nets are somewhat awkward to use; various extensions were developed providing new modelling constructs. Some of these extensions provide easier modelling but offer the same formal expressiveness as classical Petri nets, some also provide more expressiveness [7].

Petri nets are syntactically very simple; they consist of transitions and places. Transitions can only be connected to places, and places can only be connected to transitions: a Petri net is a directed (weighed) bipartite graph (a graph whose elements can be divided in two disjoint sets and whose arcs have a specified direction). The weight of an arc is a positive integer, an a k-weighted arc can be interpreted as k parallel arcs. A place can contain zero or more tokens, the assignment of tokens to places is called a *marking*. A transition is called *enabled* if in each input place as many tokens are present as the weight of the arc from this input place to the transition.

For modelling the Petri net we used CPNTools [3]. This tool models high-level Petri nets which extend classical Petri nets with hierarchy, colour and time [1], which is an extension that makes the nets more concise and readable. The tool uses a functional language 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. Defining the inscriptions tends to clutter the model but cannot be avoided in order to precisely define the execution semantics.

[3] see `http://wiki.daimi.au.dk/cpntools/`

5 WSMX execution semantics

WSMX is a event-based system, consisting of different components that communicate using events. They exhibit an asynchronous form of communication, one component raises an event with some message content and another component can at some point in time consume this event and react upon it. Figure 1 shows a simplified view of the architecture of WSMX. The *manager* is the central component responsible for invoking the different components in order to achieve the required functionality. In a usual operation WSMX is invoked by some back-end application with a specific goal to be resolved; the *manager* component responds to this request and invokes the *discovery*[4], *selector*[5], *mediator*[6] and *invoker*[7] components (one or more times) to resolve the requested goal.

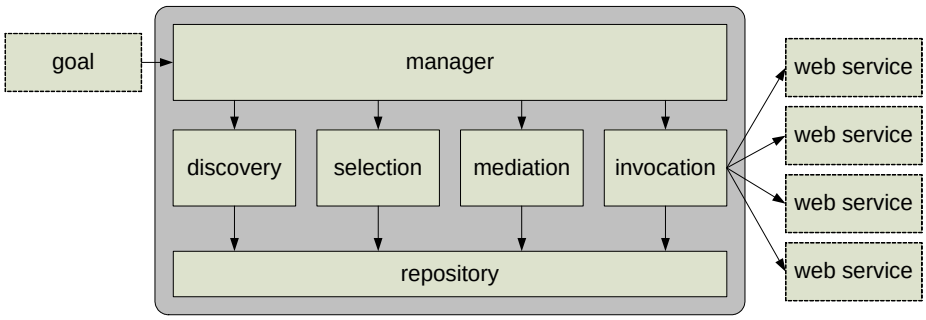

Figure 1. WSMX architecture

Figure 2 shows the execution semantics of the system for achieving a goal. This figure is hierarchical: the behaviour of some components is specified in subsequent diagrams, which is denoted by a blue rectangle underneath the transition.

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

[4] the discovery component finds services that match the requested goal.
[5] the selector selects one of the matching services.
[6] the mediator mediates data from the goal ontology into the service ontology.
[7] the invoker invokes the selected service with the mediated data.

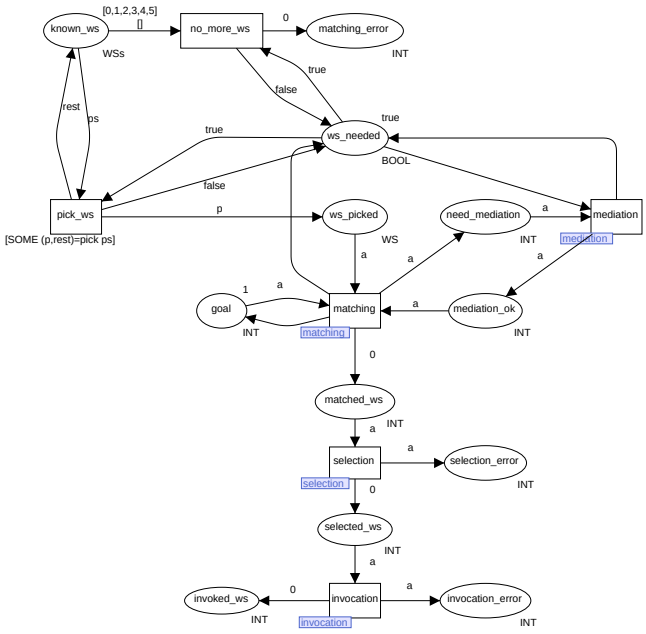

Figure 2. Execution Semantics of WSMX

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. Finally, the *invocation* component invokes the selected web service.

The process of matchmaking is specified in figure 3[8]. 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 *manager* component) a non-deterministic choice, either a match is found, or an error occurs or a mediation is needed; this choice is not made by the *manager* component but by the *matching* component. 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.

The same pattern repeats for modelling the other components, all of whose outcomes are nondeterministic exclusive-or's. Both the *mediation* component

[8] figures 3– 6 are shown in the appendix.

and the *selection* component can either fail or succeed. This is modelled in figures 4 and 5.

The *invocation* component is modelled slightly different, since this component should retry an invocation a number of times in case of network time-outs or other temporary errors. This component is shown in figure 6; 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.

6 Using execution semantics

As stated in section 1, the goal of specifying the execution semantics for WSMX was threefold: (i) to help developers understand the system, (ii) to deduce certain properties of the system and (iii) to enable model-driven execution of components. Let us investigate whether the execution semantics as specified are an adequate response to these goals.

Understandability The specification is not easily readable. Although the syntax of a Petri net is very simple and the semantics can be explained in a few sentences, most people involved in the WSMX project did not easily understand the model. It is hard to understand what has been specified exactly and the model gets easily cluttered when specifying complex systems. A result of this unreadability was a reduced interaction in specifying the model and reduced discussion about the specification. Also when implementing the system the specification was not really used as a guideline.

Secondly, it turned out to be quite hard and time-consuming to devise and maintain the specification. A significant amount of time was spend figuring out how to model certain mechanisms in the Petri net. Since system development is usually an iterative process the modelling technique should support change management; a specification that is hard to change and hard to maintain is not preferable. Problems with maintainability and readability of complex Petri nets are well-known[9].

However, it turned out that simulating the model was very useful in overcoming this problem: people tend to have difficulties visualising what a Petri net means and simulating the net clearly helps making the specification concrete. Being able to go through some example and see how the Petri net reacts helps a lot in understanding the specification. Also for provoking discussion and getting developers to comment on the model simulation proved very helpful.

Model checking On the decidability and complexity of different problems in Petri nets a lot of work has been done; the results are of course depending on the expressivity of the Petri net variant. Some common problems are checking reachability (whether some marking is reachable from some other marking), boundedness (whether the number of tokens in each place is finite) and liveness

[9] for example [2] argues that modelling certain complex patterns in Petri nets easily ends up in "spaghetti code".

(whether the system is deadlock-free) of a classic Petri net; these properties have been shown to be decidable. The subset problem (is the reachability of one net the subset of the reachability of some other) or the equivalence problem (is the reachability of one net equal to the reachability of some other net) on the other hand have been shown to be undecidable [9].

A number of tools exist that implement different model-checking algorithms (c.f. [12]). However, these tools operate on specific variants of Petri nets and can in general not be used for checking CPNTools models. CPNTools itself is able to verify certain properties of a model such as syntactical correctness, unreachable (unused) places or unsatisfiable conditions; it also allows for more complex analysis of the constructed model using state-space analysis.

In general one can prove several properties of Petri nets and tool support is available. In practise we did not use this possibility. The main reason was that (during the implementation of the system) developers had to manually translate the specification into Java code. Since the model was not easily readable the translation was not performed very strictly and resulted in an implemented system that was not equivalent to its specification. We can therefore not guarantee a relation between the modelled system and the implementation: any prove inside the model is thus useless in reality.

This is obviously not a satisfying situation, which led us to try and remove the step where developers translate the specification into programming code:

Executable specification The general idea is to use the specification as input for an execution engine. The engine should execute the model and inform components whenever they should perform a task. The step of implementing the execution semantics in programming code can be skipped: the specification itself would be executable by an engine.

In general the main argument against directly executing specifications is that it is not an efficient solution of the specification [6]. However, in component composition (which is what we try to achieve here) this might very well not be relevant: first of all the composition is quite simple and secondly efficiency of the composition is not that important as long as the components themselves are implemented efficiently.

One technical arose in this scenario: the tool we used is able to act as an execution engine for the model (it is basically the same as running a simulation of the model) but all communication to external components is synchronous. This means, one is able to specify that some transition should be executed by an external component and when this transition fires (in the model) the engine makes a call to that component and informs it to do something. This call however is synchronous, which means that the engine is waiting for this component to finish his task before executing something else in the model.

7 Related Work

We presented the execution semantics of WSMX; that work is novel and no direct related work exists. Secondly we argued for a certain application of software

specification: we want not only prove certain statements about the specification but also execute it, instead of translating it to software code.

Several approaches exist on verifying compositions of web services (which can be seen as a special case of composition of components). For instance, [8] describes how to simulate and verify compositions of OWL-S[10] web services. In this work an OWL-S process composition is translated to a Petri net, existing simulation and verification tools and techniques are then used on the Petri net. Although the paper claims to support enactment of the model no details are given. However, since the possibility of executing the specification for our approach very relevant is and since we learnt from practise that asynchronous enactment is not trivial, we would like to see in more detail how this could be done in this approach.

On the other hand, research and industry activities in business process management have lead to many techniques, languages and tools for modelling and executing processes. WS-BPEL for example is a language for modelling business processes based on web services. It could be used to model and execute composition of components, as long as those components are exposed as web services. However, to proof properties of a model the language must at least have a formal semantics, which is not the case for WS-BPEL: the interpretation of a WS-BPEL model depends on the implementation of some specific engine. There are several attempts at providing a formal semantics for WS-BPEL for example in [5]. One problem with this approach is, that it is not part of the standard itself and one is thus depending on specific engines to adhere to this semantics. In that sense the approach does not differ much from defining an operational semantics for some specific WS-BPEL engine, which is only useful if the specification is executed on that specific engine.

8 Conclusion and Future Work

Specifying the execution semantics of WSMX is part of the software development process. We have specified the execution semantics of WSMX with three objectives: to help developers understand the system, to be able to prove some properties of the model and to enable model-driven execution of components.

After providing the specification, using Petri nets, we have explained whether we think the specification actually achieved these objectives. We conclude that as presented here the specification does not meet its objectives: (i) it is not easily readable, (ii) since the model and the implementation diverge proving properties is not relevant, and (iii) component execution was technically problematic since synchronous communication is used.

We can conclude that Petri nets are not a useful technique for specifying system behaviour because they are not easily readable; one needs to use a technique that is easy to devise, maintain and use. Secondly technical problems have to be tackled when implementing model-driven components execution, which should be taken into account when selecting the modelling technique.

[10] OWL-S (formerly known as DAML-S) allows for semantic mark-up of web services.

We hope to investigate in future work whether other techniques can be used to overcome these problems. As a first step we have started using YAWL [3], a novel workflow management language. It builds on the formal foundations of Petri nets but is specifically designed for usability, which could be an advantage over a purely Petri net based approach. The system includes an enactment engine and a design tool; the system is however quite young and not yet mature. We have made some initial tests in using YAWL as component manager for WSMX; we hope to continue this work in the future.

A different approach could be to generate out of the model used for verification and enactment a different model that is solely used for clarification. One could for instance generate UML[11] diagrams out of the Petri net model, which might be more readable for developers.

Acknowledgements We would like to thank the referees for their valuable comments on a previous version of this article.

References

1. W. M. P. v. d. Aalst, K. v. Hee, and G. 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. W. M. P. v. d. Aalst and A. H. M. t. Hofstede. Workflow patterns: On the expressive power of (petri-net-based) workflow languages. In K. Jensen, editor, *Proceedings of the Fourth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2002)*, volume 560 of *DAIMI*, pages 1–20, Aug. 2002.
3. W. M. P. v. d. Aalst and A. t. Hofstede. YAWL: Yet another workflow language. Technical Report FIT-TR-2003-04, Queensland University of Technology, 2003.
4. E. Clarke and J. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, Dec. 1996.
5. R. Farahbod, U. Glässer, and M. Vajihollahi. Abstract operational semantics of the business process execution language for web services. Technical Report TR 2004-03, "School of Computing Science, Simon Fraser University", Apr. 2004.
6. A. M. Gravell and P. Henderson. Executing formal specifications need not be harmful. *IEE/BCS Software Engineering Journal*, 11(2):104–110, 1996.
7. T. Murata. Petri nets: properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr. 1989.
8. S. Narayanan and S. A. McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of 11th World Wide Web Conference*, pages 77–88, 2002.
9. J. L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, Sept. 1977.
10. C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für Instrumentelle Mathematik, Bonn, 1962.
11. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
12. E. Verbeek and W. M. P. van der Aalst. Woflan 2.0: A Petri-net-based workflow diagnosis tool. In M. Nielsen and D. Simpson, editors, *Lecture Notes in Computer Science: 21st International Conference on Application and Theory of Petri Nets*

(ICATPN 2000), Aarhus, Denmark, volume 1825, pages 475–484. Springer-Verlag, 2000.

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

Appendix A: WSMX Execution Semantics

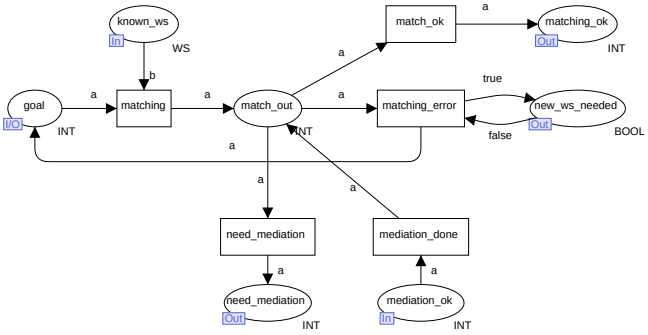

Figure 3. Matchmaking in WSMX

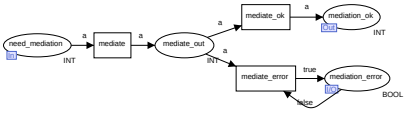

Figure 4. Mediation in WSMX

Figure 5. Selection in WSMX

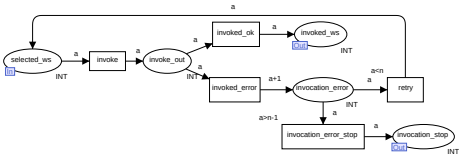

Figure 6. Invocation in WSMX