Towards Dynamic Execution Semantics in Semantic Web Services

Michal Zaremba
Digital Enterprise Research Institute
National University of Ireland, Galway
Galway, Ireland
+353 91 495009
michal.zaremba@deri.org

Christoph Bussler
Digital Enterprise Research Institute
National University of Ireland, Galway
Galway, Ireland
+353 91 492759
chris.bussler@deri.org

ABSTRACT

The research carried out on the Web Services Execution Environment (WSMX) provides a Semantic Web Services reference architecture. The design process of WSMX includes providing a formal specification of the operational behavior of the system called execution semantics. In general, the reason to formally model system behavior during the design process is to improve understanding of the system, to verify properties of the model of the designed system and to enable model-driven check-in of execution of the components. In our research on WSMX we envisioned dynamic execution semantics, i.e., a system run-time deployable formal definition of the system behavior, which can be executed against components that are part of the system. In this paper we present the dynamic execution semantics in the Semantic Web Services architecture implementation of WSMX.

Categories and Subject Descriptors

H.4 [Information System Applications]

R2.7 [Distribution, Maintenance, and Enhancement]

General Terms

Design

Keywords

Semantic Web Services, Architecture, Execution Semantics.

1. INTRODUCTION

Execution semantics, also called operational semantics, is the formal definition of the operational behavior of a software system. It describes in a formal language how the system behaves. Because the meaning of the system to the outside world consists of its execution behavior, this formal definition is called 'execution semantics' [11]. In our research on the Web Services Execution Environment (WSMX)[1] we enable dynamic execution semantics: a deployable formal definition of the operational behavior of the system which can be used against components that are part of this system. We realize it by tying together deployable (pluggable) components and Service Oriented Architecture (SOA) paradigm of business process definition. In this paper we prefer to use term execution semantics or system behavior instead of SOA term of business process, as we believe that "business process" is a too high level term that is more relevant to the behavior of several systems rather than functional software components. The dynamic execution semantics of the system allows modifying and tuning behavior of the system dynamically during run-time.

The increasing demand for faster software component development cycles combined with the desire for extending components functionality and the requirement for the components decoupling require from system designers to build adequate application configuration and management infrastructure into their systems. Without allowing for reconfiguration, management, and monitoring, software components fail to deliver to customers their full potential of usefulness and flexibility. Software systems carrying out critical operations should be able to host deployable components and to allow reconfiguring them not only during initialization, but also during runtime.

One of the characteristics of a Service Oriented Architectures (SOA) is that they are more process than component oriented. In a component oriented system, like for example an EJB[2] or CORBA[3] architecture, developers used to have objects and states that system have to manage. In process oriented systems it is more about communicating with a particular service, passing to it the required data and information and getting back results. The system does not really hold states for particular services that it talks to during process execution, while it continues to move data between components to achieve some functionality.

On the one hand side, initialization and run-time system component management is achievable these days with specifications like for example Java Management Extensions (JMX)[4]. On the other hand side, SOA delivers systems where business processes describing components interactions can be defined and executed. While many systems have been equipped with the mechanisms allowing deploying and configuring system components during its run-time, a mechanism is still missing to provide the functionality enabling deploying of the formally defined execution semantics. In existing systems the

WWW 2005, May 10--14, 2005, Chiba, Japan.

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

[2] Enterprise JavaBeans Technology - http://java.sun.com/products/ejb/

[3] Corba - http://www.corba.org/

[4] Java Management Extensions - http://www.jcp.org/aboutJava/communityprocess/final/jsr003/

reconfiguration and tuning of the behavior of the system is always taking place during coding time. Similarly to increased demands for component manageability mechanism, we recognized additional requirement for the run-time system execution semantics reconfiguration.

In this paper by dynamic execution semantics term we mean any formal abstract definition of the system behavior that can be deployed and executed on a running instance of the system. Through our research on WSMX we allow users (more specifically, administrators) of the system to formally specify new execution semantics and deploy it, delivering a completely new functionality that was not planned during system development. The research on dynamic execution semantics is carried out in the WSMX working group working in a bigger context of research on the architecture for the Semantic Web Services[5].

This paper is structured as follows. Section 2 presents execution environment for Semantic Web Services – the WSMX architecture and its entry points. Section 3 discusses how SOA can be extended to achieve dynamic execution semantics. Section 4 summarizes related work. Finally, section 5 presents our conclusions and further intentions.

2. EXECUTION ENVIRONMENT FOR SEMANTIC WEB SERVICES

In order to support distributed heterogeneous applications built by different vendors Web Service technology has been developed. Existing Web Service cornerstone technologies such as UDDI [3], WSDL [2] and SOAP [9] provide the basic functionality for discovering (UDDI), describing Web Service interfaces (WSDL) and exchanging messages (SOAP) in heterogeneous, autonomous and distributed systems. In practical terms, existing Web Services support operations which are limited to independent calls over a network, fixed collaboration patterns or predefined supply chains. Web Service technologies and standards do not provide any functionality to specify how to include additional semantic information which would allow using and processing them without any human interactions.

The approach to systems integration based on semantically enhanced Web Services is nowadays possible through Semantic Web Services. The Web Services Modeling Ontology (WSMO)[6] working group is one of the few research efforts developing a conceptual model, language and execution environment for Semantic Web Services (SWSs). Enhancing existing Web Service standards with semantic markup is standardized through the WSMO working group and promotes already existing Web Services standards for semantic-enabled integration. Semantic markup is exploited to automate the tasks of service discovery, composition, invocation and interoperation enabling seamless interoperation between them [4] and keeping human interaction to minimum.

2.1 Reference Architecture for Semantic Web Services Infrastructure

The Web Services Execution Environment (WSMX) is one of three WSMO working groups that provide an execution environment called WSMX. WSMX enables discovery, selection, mediation, invocation and interoperation of SWSs. The goal of the research on WSMX is to provide a reference architecture for Semantic Web Service systems. WSMX is based on the conceptual model provided by the Web Services Modeling Ontology [14] which describes all aspects related to SWSs. The mission of the WSMX working group is to define a Semantic Web Services architecture and to provide a complete implementation of WSMO. WSMX is a reference implementation for WSMO providing the proof of its applicability and usefulness as well as being a vehicle for driving new projects and partnerships. The goal is to provide both a test bed for WSMO and to demonstrate the viability of using WSMO as a model for achieving dynamic interoperation of SWSs.

The development process for WSMX includes defining its conceptual model (the WSMO ontology), modeling its execution semantics (processed by dynamic execution semantic engine capable of interpreting formal definition of system behavior) and designing the WSMX system architecture. The WSMX working group also defines component interfaces, designs particular components and provides their implementation. The research through WSMX working group provide guidelines and justification for a general SWS architecture, while at the same time development team provides reference implementation of the system[7] [10].

Apart from the cornerstone SWS functionalities that must be available with any execution environment for SWSs, such as discovery, mediation or invocation the working group also addresses some more specific system functionalities while developing the WSMX system (although they remain out of scope of WSMO itself, as WSMO is only concerned with the external behavior of Semantic Web Services). One of these additional features, which are currently under development in WSMX is a dynamic execution semantics.

2.2 WSMX as SOA

WSMX is a Service Oriented Architecture (SOA), which means that it is a software system consisting of a set of collaborating software components with well defined interfaces that together perform a task. These components do not necessarily execute in the same address space, not even necessarily in different address spaces on the same machine. Instead, they may very well execute on different network locations communicating over a network connections through multiple protocols. This situation creates its own unique demands, namely latency, memory access, concurrency and failure of subsystems that the architecture must be able to cope with. All these aspects are addressed in subsequent phases while designing and implementing WSMX. SOAs differentiate themselves from other distributed systems through the concept of loose coupling brought to its extremes. Strong decoupling of the various components, which realize an e-commerce application is also one of two major features of WSMO. In WSMX, conceptually independent pieces of functionality have been grouped into components. Each of the WSMX components provides services – each of which is a logical unit of system code, application code, and persistence layers, in short, anything that as a unit can carry out an operation. Services are often characterized by

[5] http://www.wsmx.org

[6] Web Services Modeling Ontology (WSMO) – http://www.wsmo.org

[7] WSMX at sourceforge – http://sourceforge.net/projects/wsmx

exactly these operations that they provide to other components. The WSMX architecture and its loosely coupled components are presented in Figure 1.

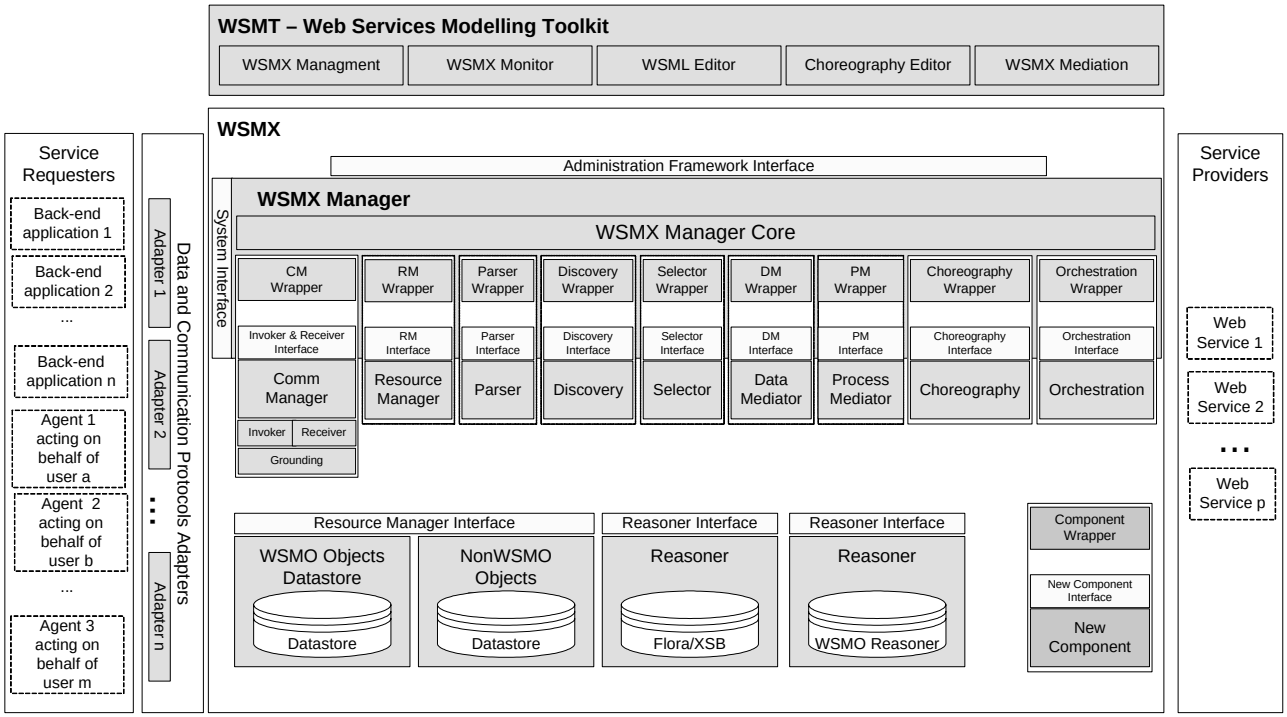

Figure 1. Architecture for Semantic Web Services

For each component a separate public interface has been defined, that is known by all other components provided with the reference implementation or by components provided by independent components providers. Components can be plugged-in and plugged-out from the system both at startup as well as during run-time. The components that are provided with the WSMX reference implementation can be easily replaced by other components, e.g., those provided by third parties. At this stage we are still working with the standardized infomodel for system interfaces, because we depend on one of design principles of SOA, which demands separation of interface from the implementation (and that is why it requires to known interfaces before system execution). To fully achieve dynamic execution semantics we must be able to deploy interfaces to running system, so both new components with new interfaces and new executions semantics can be defined and executed. This aspect has been not yet addressed.

The WSMX reference implementation provides the complete implementation of all of the components and users of WSMX may still decide to use components delivered by other providers if they chose to prefer those.

In distributed SOA systems the communication between components is taking place through events. Although we also recommend building the communication inside the Semantic Web Services architecture on the event paradigm, the system components can be coordinated without events. It enables the best practice of implementation abstraction through interfaces, by which the implementation of a service must be of no concern to the client of the service. All of this together results in increased flexibility, better extensibility and dramatically improved reusability. However, even if the right architectural decisions are taken, it is not always easy to achieve all of these goals simultaneously. Scalability and proper service structuring are crucial and have to be taken into account, too.

2.3 System External Behavior

Currently defined functionality provided by the WSMX system as a whole can be described in terms of its entry points. Entry points are the standardized interfaces of WSMX enabling the communication with any external entities requesting services. More details on system functionality can be found in an execution semantics document [16], that formally specify the desired operational behavior of WSMX, serving not only as a reference for developers but also as a means of validation and model-checking. While execution semantic defined through WSMX working group remains static, dynamic execution semantics mechanism enables extending behavior of the system and adding new entry points with each new execution semantics deployed.

In the current version of the WSMX system we define one formal execution semantics with four possible branches, each of them starting with one entry point. These four entry points must be available in each working instance of system, which is WSMX compliant. Entry points also define the required functionality of any WSMX compliant system. By selecting a given entry point the predefined execution semantics is triggered. These four obligatory entry points enable the execution of any of four available execution semantics:

```
realiseGoal(Goal,
OntologyInstance):Confirmation
```

Any external entity which expects to get its goal realized without back and forth interactions (communication) with

WSMX system might wish to provide a formal description of a Goal (in WSMO terms) and an instance of Ontology (some data required for processing of a Web Service). This quite simplified scenario assumes that service requester knows even before the service discovery all the data that might be required by the service provider. WSMX selects and executes the Web Service on behalf of service requester. The service requester might receive a final Confirmation, but this step is not obligatory (many entities that might wish their goals to be realized by WSMX system might not have permanent addressing, so there is no possibility to make an asynchronous call back to them returning the final result of the service invocation).

```
receiveGoal(Goal):WebService[]
```

The `receiveGoal` entry point addresses the scenario when a service requester consults WSMX to learn about Web Services that satisfy its Goal. In this asynchronous call the service requester provides a Goal and expects to get back a set of Web Services.

```
receiveMessage(OntologyInstance,
WebServiceID, ChoreographyID):Confirmation
```

Once a service requester knows the Web Service that he wants to use, it must carry back and forth a conversation with the WSMX system in order to provide all the necessary data to execute this Web Service. By giving fragments of Ontology Instances (e.g. business documents such as Catalogue Items or Purchase Orders in a given ontology) and a reference to the Web Service and Choreography (only if choreography has been instantiated already) that is to be used, it provides all data required by Web Service of service provider.

```
storeEntity(WSMOEntity):Confirmation
```

The StoreEntity entry point provides an administration interface for the system enabling to store any WSMO related entities (like Web Services, Goals, Ontologies) and to make them available for other users of the WSMX system.

In addition to these four entry points we assume that each instance of a Semantic Web Services system provides an engine to support dynamic execution semantics enabling execution of any formal description of system behavior. In this way we can also define additional functionality.

3. REALISING DYNAMIC EXECUTION SEMANTICS

Service Oriented Architectures have weaknesses that can be overcome by a dynamic execution semantics functionality. The following subsections will discuss these weaknesses and show how dynamic execution semantics can be applied for any distributed system.

3.1 Advantages and Weaknesses of Service Oriented Architectures

Service Oriented Architecture (SOA) is a new architectural paradigm enabling integration of heterogonous applications by decoupling functionality provided by distributed software components. Systems which are going to use this functionality bind components at a run-time allowing them to cooperate on the process level.

SOA is an architectural paradigm in which two computing entities interact in such a way that one entity is able to perform requested tasks on behalf of another entity. Traditionally, achieving a complete functionality of a given system requires many building blocks to be integrated together in one complete software package. Such an approach usually delivers very monolithic software, where code to accomplish integration of application functions is mixed together with code performing these functions. In SOA these building blocks become independent software applications offering their functionality in form of services to any entity (software system), which requires them. A requesting party is capable to use this functionality without knowing the details of this service internal implementations, because the whole functionality is offered through the well defined interfaces (in this moment mostly Web Services standards are used for this purpose, but also XML over HTTP and JMS may be used to achieve similar results [5]). To meet demands of SOAs, software packages must be modularized in order to enable functionality reusability across various systems. The SOA approach to architecture enables interoperability between systems and platforms, which have been designed and implemented independently from each other, use different programming languages, operating system, hardware configurations etc. Well defined interfaces of components allow for an interaction like style of communication between particular software entities.

SOA allows software systems to be more agile and more responsive by adapting more quickly to changing business needs. With the growing importance of Internet and e-business style of interaction between companies, the SOA approach to architecture offers a promising paradigm to integrate existing back-end application systems and separated processes across the whole value chains of companies from suppliers to final customers of the product or service.

The development of a traditional system usually follows the four steps conventional path of requirements analysis, designing software, implementing and testing. Applying formal methods during software design such as modeling formal execution semantics improves the result of the design allowing verifying a software system before it is build. Having a formal definition of the system behavior, the architecture can be designed and implement ensuring that no design flaws, livelocks or deadlocks will be ever encountered during system execution. The formal definition of execution semantics allows revealing ambiguity, incompleteness and inconsistency before the actual system implementation [15]. Although the traditional approach to system development enhances developers' understanding of the system before it is implemented, the new requirements for more agile and adaptive architectures demand from system designers to deliver systems where flexible execution semantics can be defined and deployed on the existing system without the need to rewrite once written core system code. In mission critical applications which have to run 24/7 it might be not even feasible to shut down the system, while new executions semantics is deployed.

In a component-based system different components must cooperate together to achieve the required functionality of the system. In SOA architecture to make this coordination a reality, software developers usually provide a central control component, which requests functionality from other components. The main task of this management component is to control and coordinate the execution of other components. Usually it is responsible for providing all the necessary data required by components to fulfill its tasks. The control flow

among components of the system remains hard-coded into management component. Every time the execution semantics of the system is getting updated the management component has to be reprogrammed and recompiled.

In distributed systems, where components might be provided by different vendors the whole situation becomes even more complicated. Including one additional component or even changing only an interface of existing component requires the whole system to be formally verified again. Management component might have no influence on a provider of a component. That is why it becomes necessary to formally verify if new execution semantics defined for this system can be executed with the given set of components. It should be possible to describe the semantic requirements of a management component and to match it with the semantic capabilities of other components. We do not claim here that such matching should happen any time a process is going to be executed, but only when a new execution semantics definition is available and should be deployed on a given infrastructure. From the list of known components, the management component should only connect the ones that are capable to this request.

Based on the decoupling requirement of SOA systems, the functionality of platforms that are based on these concept remains restricted. The system focuses on supporting process-service binding mechanism to achieve functionality offered by distributed components. We perceive the current potential of SOA architectures still being inadequate to address the requirements of Semantic Web Services systems. In order to enable fully agile and responsive systems adapting to requirements of service requesters and providers, we envision that the execution semantics of such a platform should be adaptable to particular scenarios, which remains unpredictable during design time of the system. While SOA allows on modifying business processes to achieve some functionality, it remains silent on dynamic deployment of such a process in running instance of the system. To achieve dynamic execution semantics we propose to extend SOA by allowing for model-driven execution of system components. Such execution semantics must be not only able to be applied at the design or startup time of the system, but must be executable in a running instance of the system as well. We apply and test this approach to execution semantics in our reference implementation of Semantic Web Services architecture.

3.2 Use Case for Dynamic Execution Semantics

The first WSMX implementation included only one possible execution semantic which was hard-coded with components of the system. In the second implementation we already included four possible branches which can be executed during run time of the system. Subsequent progress on the system development is possible by revealing new requirements and use cases provided by potential users of the system. As these use cases remain confidential and cannot be presented in this paper a similar use case is shown to picture the requirements for the dynamic execution semantics.

The potential application used by the dealer for several insurance companies automates the process of collecting insurance information from them, taking into account that the data can be accessed by executing Semantic Web Services. This aggregated information is provided to users, according to the data that they have filled-in in appropriate query forms. While company can use such components like data or process mediator from any component provider, it does care that discovery engine is hosted locally by them and only this particular discovery engine is used to provide list of available services. The company also wants to make sure that nobody else except them can use this discovery engine. Additionally to that they like to provide themselves an additional component used to carry out the evaluation process of some of the quotes properties and modifying them before they are returned to user (e.g. adding they own margin on top of quote returned by insurance company). With dynamic execution semantics as presented in this paper, we can restrict process execution to particular set or even individual components (by creating new types of events that can be only consumed by these components) and we can easily add new components, which were not considered during design time of the system.

3.3 Dynamic Execution Semantics Engine

As mentioned before, during the design process of WSMX several steps have been undertaken, including describing system conceptual model, specifying the formal system execution semantics and defining and designing its architecture. By providing the conceptual model the common reference vocabulary for the development team has been defined, which has been used during different phases of the project. Execution semantics, the formal specification of the operational behavior is normally used for a number of reasons during software development. As described in [16] in the context of WSMX we were initially interested in modeling the execution semantics of WSMX in such a way that

(1) developers can understand the system themselves,

(2) certain properties of the system can be inferred,

(3) model-driven execution of the system's components can be enabled.

The architecture definition has delivered a detailed description of the overall system, components interfaces and specification of the functionality expected from particular components.

One of the key design decisions we undertake for WSMX has been to keep individual components decoupled from each other and to enable components distribution across the network. The development of the high-speed networks makes it possible to distribute services across various machines achieving complete functionality by combining these partitioned and distributed resources. The effective way to exploit advantages provided by network and achieve system scalability requires partition system functionality into coarse-grained components with well defined interfaces which can provide a small but complete functionality required by this system.

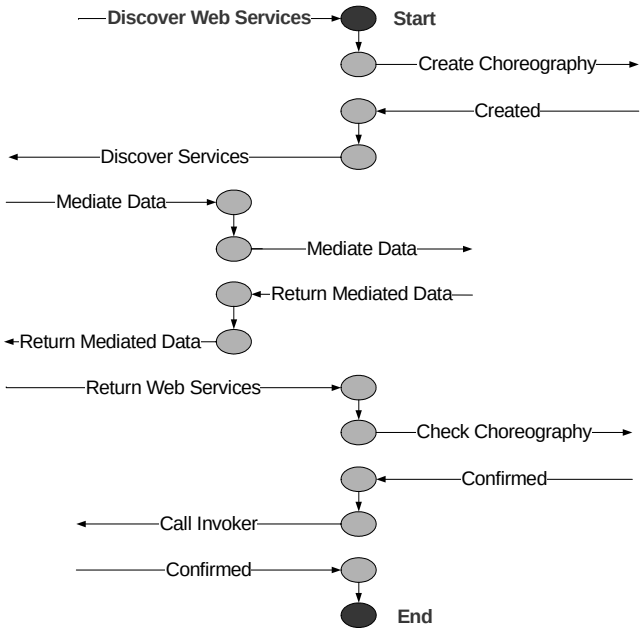

Figure 2. Define execution semantics

An initial version of WSMX included only one possible execution semantic which was hard-coded into components of the system. This approach very quickly showed the weaknesses of the WSMX design as revealed by new requirements and use cases provided by potential users of the system. We decided to refine our initial architecture approach and we accommodated a mechanism to incorporate new components and methods enabling adding and removing any new formal definitions of execution semantics describing operational behavior of the system without the need to recompile the whole platform any time such new execution semantic becomes available.

The first step to enable new execution semantics in the system requires to design the execution semantics branch (or in SOA terminology - "business processes") grouping business activities in pursuit of a common goal. While for WSMX we used Petri nets [1], we have not restricted dynamic execution engine to any specific formalism. The tool we are using, CPNTools [13], makes it possible to model so-called high-level Petri-nets, extending classical Petri nets with hierarchy, color and time. The tool used for definition of business process must make it possible to verify certain properties of the model; it must check some simple properties such as syntactical correctness, unreachable states or unsatisfiable conditions. While we compose a process from a set of components, we only make an abstract declaration that a service might be requested from the component by using known service interface from the system infomodel (see figure 2).

Actually we do not bind the process to the concrete service, which is going to be invoked during run-time. As an underlying formalism to define abstract business processes in WSMX 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, but the system itself is however quite young and not yet mature. We have already made some initial tests in using YAWL as system behavior definition formalism for WSMX.

Besides process definition, we recognized another requirement to enable dynamic execution semantics: the ability to plug-in and plug-out components at runtime. In WSMX we enable reconfiguration, management, and monitoring of available software components. We maintain that the system must be capable to host deployable components and to reconfigure them during initialization and as during runtime (see listing 1 for system configuration file definition). This configuration mechanism is used during system start-up to pick up all known components. Additionally to it, during run time new components can be added and old components can be removed.

```
<?xml version="1.0" encoding="UTF-8"?>
<sc:wsmxconfiguration
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:sc="http://www.wsmx.org/simpleconfiguration"
                    systemcodebase="/path/to/distribution"
                    port="9000">
    <domain name="components">
        <mbean name="Discovery"
               class="ie.deri.wsmx.discovery.wrapper.Discovery"
               componentcodebase="./discovery.wsmx"
               eventtypes="resourcemanager.response,
                           wsmx.discovery.wsmlmessage.validated">
            <property name="vendor">
                wsmx.org
            </property>
        </mbean>
        <mbean name="ResourceManager"

class="ie.deri.wsmx.dbManager.wrapper.ResourceManager"
               componentcodebase="./resourcemanager.wsmx"

eventtypes="wsmx.registration.wsmlmessage.validated">
            <property name="vendor">
                wsmx.org
            </property>
        </mbean>
        <mbean name="CommunicationManager"
```

```
class="ie.deri.wsmx.communicationmanager.wrapper.CommunicationMan
ager"
               componentcodebase="./communicationmanager.wsmx"

eventtypes="wsmx.registration.wsmlmessage.persisted">
            <property name="vendor">
                wsmx.org
            </property>
        </mbean>
        <mbean name="Mediator"
               class="ie.deri.wsmx.mediator.wrapper.Mediator"
               componentcodebase="./mediator.wsmx"
               eventtypes="wsmx.void">
            <property name="vendor">
                wsmx.org
            </property>
        </mbean>
        <mbean name="Choreography"

class="ie.deri.wsmx.choreography.wrapper.Choreography"
               componentcodebase="./choreography.wsmx"
               eventtypes="wsmx.void">
            <property name="vendor">
                wsmx.org
            </property>
        </mbean>
        <mbean name="Parser"
               class="ie.deri.wsmx.parser.wrapper.Parser"
               componentcodebase="./parser.wsmx"

eventtypes="wsmx.registration.wsmlmessage.nonvalidated,

wsmx.discovery.wsmlmessage.nonvalidated,

wsmx.invocation.wsmlmessage.nonvalidated">
            <property name="vendor">
                wsmx.org
            </property>
        </mbean>
    </domain>
</sc:wsmxconfiguration>
```

Listing 1. WSMX components configuration

The persistent configuration support is responsible for loading the various components into memory at startup time. Different versions of WSMX provide different degrees of configuration support, early releases provide only basic, centralized support while later releases are planned to have more sophisticated and decentralized configuration systems to provide more flexibility. The system must be able to cope with the additional complexity caused by the introduction of features such as component injection or persistence of metadata objects.

Having an abstract process definition and components installed in the system, we generate wrappers for components (see figure 3). The purpose of the wrapper is to separate components from transport layer for events. As mentioned before, although we also recommend building the communication inside Semantic Web Services architecture based on event paradigm, the system components can be coordinated without events. WSMX is an event-based system, consisting of many wrappers that communicate using events. Wrappers utilze an asynchronous form of communication. One wrapper raises an event with some message content and another wrapper can at some point in time consume this event and react upon it. Components offering services to WSMX remain unaware of event infrastructure, while they solely communicate with their own wrappers, while event consumptions and production is taking place only on a wrapper level. Transport mechanism has been also decoupled from the system by using Transport interface, which hides details of event transport mechanism.

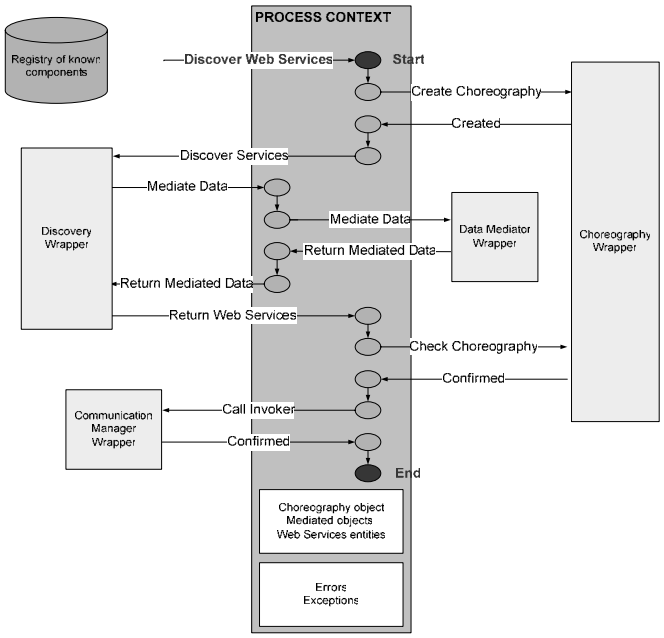

Figure 3. Creating components wrappers and instantiating process context

WSMX is an improved SOA, what means that it is a system composed of a set of distributed, loosely coupled components, but without centrally organized "business process" management unit. System components do not cooperate based on hard-wired bindings like in traditional systems, but the communication is based on events which are carried in by the transportation layer. That is, if some functionality is required from the other component, then event that represents request is created and published. Components, which wish to process given types of requests, subscribe to given event types, so whenever new events appear in a transport layer, they might be picked up, processed and consumed. In our current approach events exchange is conducted via transportation layer, which is based on Tuple Space [8], but is going to be changed in the future to Triple Space [7] mechanism. We enable seamless interaction between components without direct events exchange between them. Interaction is carried out like in the messaging systems by exploiting publish-subscribe mechanism.

Figure 4 presents the complete architecture enabling decoupling of components from execution semantics in the Semantic Web Services architecture. The deployment of any new execution semantics will regenerate wrappers for the set of components. Based on execution semantics definition, these wrappers will be only capable to consume and produce particular types of events. In a running system, the dynamic execution semantics is achieved by mapping abstract system behavior into real event infrastructure of the system.

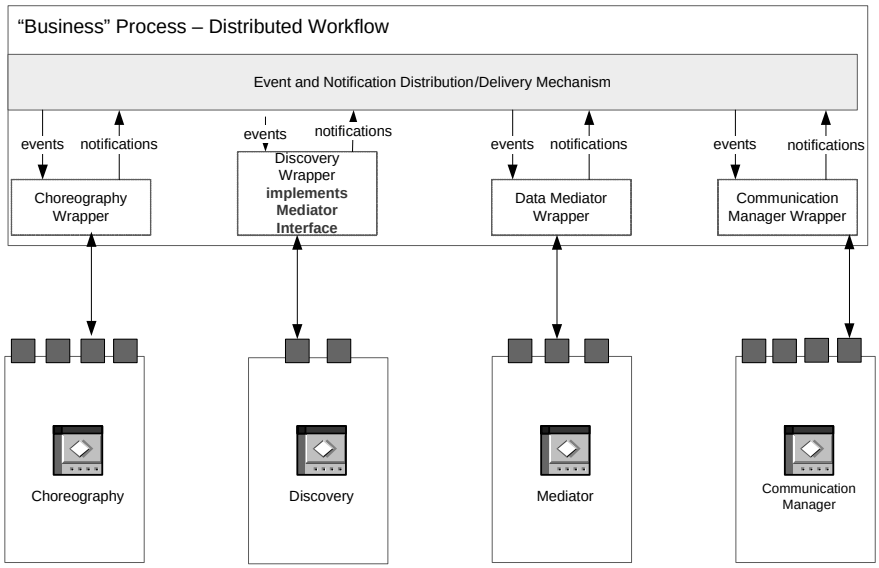

Figure 4. Realisation of dynamic execution semantics in Semantic Web Services architecture

To assure that there are no design flaws (like livelocks or deadlocks) the formal verification of the model must take place before model is deployed to the system.

4. RELATED WORK

Beside WSMX there are other software tools providing support for execution of Semantic Web Services having their roots in OWL-S, Meteor-S and WSMO initiatives. There are also several commercial integration platforms capable to overcome integration problems between heterogeneous systems. While none of them offers dynamic execution semantics, their functionality remains similar to functionality provided by WSMX, however, just on a syntactical level. In this section we provide a short overview of them.

IRS III is a platform developed by the Knowledge Media Institute at the Open University, capable of handling WSMO and OWL-S based Semantic Web Services [6]. In the IRS III design environment a provider of a service creates a WSMO based service description and publishes it against its service on the IRS III server. Having the service available, a goal can be described and bound with existing Web Service using a mediator. Although this approach sounds quite limiting as Web Services must be known to IRS III server at design time – it is different to WSMX when they are not known until run time. An attempt has been already undertaken to bring interoperability between WSMX and IRS III - currently two major WSMO compliant Semantic Web Services platforms.

Meteor-S builds on existing Web Services technologies providing a framework for Web Services composition and discovery [12]. Meteor-S is based on a language predecessor of OWL-S, which is used by OWL-S. There is no comprehensive strategy for development of Meteor-S server like in the case of WSMX or IRS III. Rather there are multiply efforts to address different aspects of Semantic Web Services. While Meteor-S tools are equal to WSMX components, its hard to talk about any execution semantics of Meteor-S as no system like in case of WSMX really exist. The main tool called MWSAF provides an ontology store, a translator a library and a matcher library. Another tool Meteor-S WSDI is a peer-to-peer infrastructure for accessing and annotating multiple registries.

OWL-S [4] is a comparable effort to WSMO initiative attempting to define an ontology for Semantic Web Services. Similarly to Meteor-S there are multiply tools available, but there is no integrated strategy regarding the development of a complete infrastructure for execution of OWL-S Web Services. There is a composer, matchmaker or editor, but a run-time infrastructure capable to handle execution and coordination between all these components is not yet available. There are some related efforts to WSMX to build OWL-S virtual machine and Mindswap's OWL-S API which can be used to develop and execute OWL-S services, but particular OWL-S tools are not yet "coupled" with this infrastructure. As there is no complete OWL-S infrastructure with all the tools connecting to it, so consequently no dynamic execution semantics can be defined for it and executed.

Besides these efforts, one can imagine building system with similar functionality offered by WSMX out of components using existing commercial integration platforms such as for example BizTalk Server 2004[8], WebSphere Integration Suite[9], Application Server 10g[10] and others. Currently none of these tools supports semantic annotations and none of them allows mediating between ontology instances. While most of them is

[8] BizTalk Server - http://www.microsoft.com/biztalk/

[9] WebSphere Integration Suite - http://www-306.ibm.com/software/websphere/

[10] Application Server 10g[10]- http://www.oracle.com/appserve

modularized and new components can be added easily (even during run time), none of them really address issue of dynamic execution semantics.

5. CONCLUSIONS

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. In this paper we present our initial approach to dynamic executing semantics aiming at designing such an infrastructure for WSMX system. While this is a big step towards enabling flexible definition of system behavior, there are still some aspects that need to be addressed. Currently components interfaces are still defined solely by java interfaces. To provide fully enabled dynamic executions semantics as envisioned in this paper, preferably services provided by components will have descriptions in machine-processable meta-data.

6. ACKNOWLEDGMENTS

This work is supported by the SFI (Science Foundation Ireland) under the DERI-Lion project and by the European Commission under the projects DIP, Knowledge Web, SEKT, SWWS, and Esperonto, and by the Vienna city government under the CoOperate program. The authors thank all members of the WSMX (http://www.wsmx.org/) working group for fruitful discussions on this document.

7. 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 *Proceedings of the second Workshop on Computer-Supported Cooperative Work, Petri nets and related formalisms*, (1994).

[2] Erik Christensen, Francisco Curbera, Greg Meredith and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1, 2001. http://www.w3.org/TR/wsdl

[3] Luc Clement, Andrew Hately, Claus Von Riegen and Tony Rogers. UDDI version 3.0, 2004. http://uddi.org/pubs/uddi_v3.htm

[4] The Owl Service Coalition. OWL-S 1.1 beta release, 2004. http://www.daml.org/services/owl-s/1.1B

[5] Mark Colan. You Have Critical SOA Questions? We have Answers!, ebizQ, 2004. http://www.ebizq.net

[6] John Domingue, Liliana Cabral, Farshad Hakimpour, Denilson Sell and Enrico Motta, IRS-III: A Platform and Infrastructure for Creating WSMO-based Semantic Web Services. in *WIW 2004, WSMO Implementation Workshop 2004*, (2004).

[7] Dieter Fensel, Triple-based Computing. in *Semantic Web Services: Preparing to Meet the World of Business Applications a workshop at the International Semantic Web Conference (ISWC2004)*, (Japan, 2004).

[8] David Gerlernter *Mirror Worlds*. Oxford University Press, 1992.

[9] Xml Protocol Working Group. SOAP version 1.2, 2003.

[10] Michal Zaremba Matthew Moran, Adrian Mocan and Christoph Bussler, Using WSMX to bind Requester & Provider at Runtime when Executing Semantic Web Services. in *WIW 2004 WSMO Implementation Workshop 2004*, (2004).

[11] Eyal Oren, WSMX Execution Semantics - Executable Software Specification. in *WIW 2004 WSMO Implementation Workshop 2004*, (Frankfurt, 2004).

[12] A. Patil, S. Oundhakar, A. Sheth and K. Verma, Semantic web services: Meteor-S Web Service Annotation Framework. in *13th International Conference on World Wide Web*, (2004).

[13] 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 *24th International Conference, ICATPN 2003*, (2003), 450-462.

[14] Dumitru Roman, Holger Lausen and Uwe Keller. Web Service Modeling Ontology (WSMO), 2004. http://www.wsmo.org/2004/d2/v1.0/

[15] Jeanette Wing A Specifier's Introduction to Formal Methods. *IEEE Computer*, *23*. 8-26.

[16] Maciej Zaremba and Eyal Oren. WSMX Execution Semantics, 2004. http://www.wsmo.org/2005/d13/d13.2/v0.2/