Enabling Components Management and Dynamic Execution Semantic in WSMX

Thomas Haselwanter, Maciej Zaremba and Michal Zaremba

Digital Enterprise Research Institute (DERI),
National University of Ireland, Galway, Ireland
University of Innsbruck, Austria
{firstname.lastname}@deri.org

Abstract. The Web Service Modelling Execution Environment is a platform for dynamic discovery, mediation and invocation of Semantic Web Services. We believe that Semantic Web Services systems should support a dynamic execution semantics, i.e., a system run-time deployable formal definition of the system behavior, which can be executed against components that are available for this system. In this paper we present the work we have completed to date on dynamic execution semantics in the context of Semantic Web Services architecture implementation of WSMX. We also document the design rationale of a microkernel and a distribution architecture for this system.

1 Introduction

Web Services Execution Environment (WSMX) is a run-time environment and a test bed for WSMO[7]. Our research aims to assess the viability of WSMO and to provide a reference implementation of the system. WSMX is composed of loosely-coupled components that carry out various tasks related to WSMO. Some of the main components of WSMX are Service Discovery, Data Mediation, Process Mediation, Service Selection, and Communication Manager. Implementation of these components is not prescriptive; however the implementation has to conform to the well-defined public interfaces of these WSMX components. This approach facilitates creation of new component implementations by third parties and fosters WSMX proliferation.

In this paper we present WSMX as the a system composed of distributed components. In our research on WSMX 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. Through our research on WSMX we allow administrators of the system to formally specify and deploy new execution semantics, delivering a completely new functionality that was not planned during system development. This paper also discusses communication paradigms among components and presents a system management approach.

This document is structured as follows: In Section 2 a short description of the WSMO and WSMX is given. Section 3 presents underlying concepts and approach to WSMX Distributed Architecture composed of set of loosely-coupled

and distributed components. Section 4 describes Components Management issues. Section 5 provides a description of communication among components via intermediary layer of *Wrappers*. Finally, related work is presented and conclusive comments are given.

2 WSMO and WSMX

Research on Semantic Web Services aims to improve systems integration based on semantically enhanced Web Services. The Web Services Modeling Ontology (WSMO) [1] 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.

The Web Services Execution Environment (WSMX) [2] working group aims to provide guidelines and justification for an architecture for the SWS systems and to design and implement an execution environment which enables discovery, selection, mediation, invocation and interoperation of Semantic Web Services (SWS). The development process for WSMX includes establishing a conceptual model, defining its execution semantics, developing the architecture of the system, designing the software and building a working implementation of the system. The research on dynamic execution semantics is carried out in the WSMX working group working in the wider context of research on the architecture for the Semantic Web Services.

3 WSMX Distributed Architecture.

WSMX is a Service Oriented Architecture (SOA), what means that it is a system composed of a set of distributed, loosely coupled components. There are no hard-wired bindings between these components; communication is based on events. That is, if some functionality is required then an event that represents the request is created and published. One of the other components subscribed to this event type can fetch and process this event. The events approach allows asynchronous communication mean, i.e. components do not block in awaiting for response. Events exchange is conducted via a Tuple Space, that provides persistent shared space enabling seamless interaction between parties without direct events exchange between them. Interaction is carried out by exploiting a publish-subscribe mechanism.

[1] Web Services Modeling Ontology (WSMO) - http://www.wsmo.org
[2] Web Services Execution Environment (WSMX) - http://www.wsmx.org

Each component provides a logical unit of functionality like discovery, data mediation, process mediation, invocation, persistence layer and so on. Component interfaces are publicly specified, therefore it is known prior to the run-time execution how to interact with each component, i.e. what operations are provided by the component. Public interfaces of a component are implementation independent. Since only interfaces are prescriptive, new implementations of components can be provided by third parties by adhering to these public interfaces and these can be plugged-in to the system. Components can be perceived as a black-boxes, i.e. their internal logic is not visible; only their external interfaces are exposed.

Components might carry out their tasks better in distributed environments (e.g. by having access to more reliable data or to more powerful machines). Generally, in a case of acquisitive tasks distribution greatly improves system scalability. Component distribution increases involvement from third parties in system development, since it facilitates add-ons to the system and allows the processing of additional tasks. Therefore, it is worthwhile to enable distributed computing for the system as a whole. Data related to a particular system context can be conveyed between remotely deployed components. Moreover, system logic can be portable as well in order to avoid centralizing parts of system, thus increasing system reliability and fault tolerance.

To enable distributed computing, an intermediary layer of *Wrappers* is introduced. Figure 1 illustrates this distribution strategy. *Wrappers* separate component implementation from communication issues and enable invocations of functionality provided by other components. This approach takes off the burden of communicating with other components via a Tuple Space from component developers. The components themselves are not aware of their distribution. From their point of view it is assumed that communication with other components is already provided and is transparent. Such an approach is more capable of adapting flexibility to changes. For example, if the the underlying communication paradigm were changed, only the *Wrappers* would be affected by the change, whilst component implementation would not require any changes at all. Since *Wrappers* are generic, there is no need to create dedicated wrappers for each type of component. Instead, each *Wrapper* representing a particular component implementation is instantiated with an event type appropriate for given component.

System behaviour is described by its execution semantic. Since WSMX consists of loosely-coupled components, various execution semantics can be built by combining existing components and by specifying interactions between them. There are four mandatory execution semantics for each instance of the WSMX system[9], but new execution semantics can be created and fed into WSMX. Moreover, thanks to its architecture WSMX can be easily enhanced with new components. Components can be dynamically plugged in or plugged out. New versions of components can replace outdated ones in this manner. This gives the designer a flexible way to create new execution semantics.

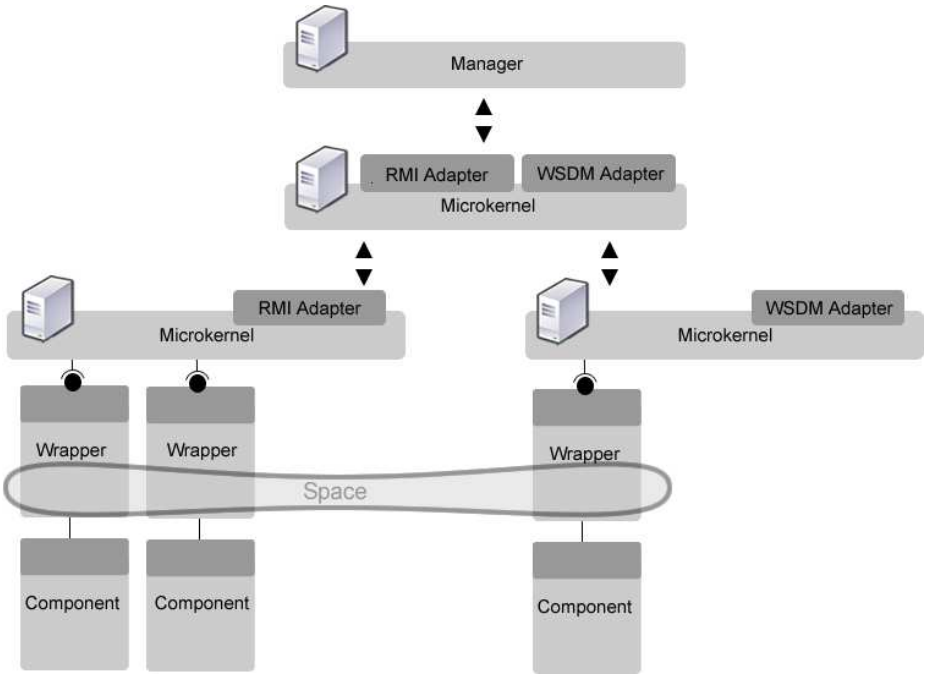

Fig. 1. Distribution in WSMX

4 WSMX Core - Components Management

As in all systems of a certain complexity, management becomes a critical issue. In WSMX we make a clear separation between business logic and management logic, treating them as orthogonal concepts. If we did not separate these two logic elements, it would be increasingly difficult to maintain the system and keep it flexible. From a certain perspective it could be argued that the very process of making management explicit, captures another invariant that helps to leverage the support for dynamic, informed change of the rest of the system. Figure 2 provides an overview of the WSMX management framework.

4.1 Microkernel

The WSMX microkernel is a management agent that offers several dedicated services, the most essential of which is perhaps the bootstrap service, responsible for loading and configuring the application components. The necessary information is obtained by a combination of reflection and supplied information in the form of a distributed configuration. It is possible to achieve hot-deployment of a component which is done by copying a "wsmx" archive (a jar-like archive with the standardized structure) to the system codebase. The kernel will detect the archive, load the contained resources as bytecode, define the classes, inject

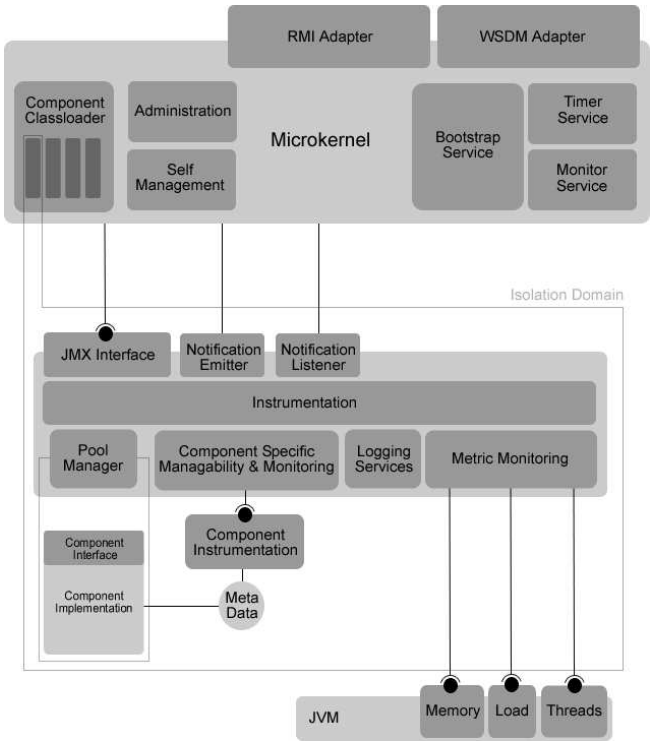

Fig. 2. WSMX Management Framework

a wrapped component instance into the running WSMX and connect it to the space. A precondition to the decision to develop microkernel was the opportunity to design the system from scratch. Once the infrastructure is in place this decision will contribute to the reliability of the whole system since. The agent plays the role of a *driver*. It is built into the application, as opposed to a *daemon* approach, where the agent would operate in its own JVM, alongside components that execute in a different JVM.

The kernel may employ self-management techniques such as scheduled operations, and additionally allows administration through management consoles by operators. *Management consoles* may be anything from command prompts or web interfaces to dedicated standalone management applications. While the former are better embedded in WSMX the later connect through Connectors or Adapters that operate on a specific protocol such as RMI. Of particular interest is a Web Services Distributed Management (WSDM)[5] compliant exposure, since it operates in a platform independent way that can be accessed by the most different types of clients.

4.2 Management Aspects of Wrappers and Components

As mentioned above, *Wrappers* are host to a number of subsystems that provide services to components and enable communication with other components. Besides *Revivers* and *Proxies*, which are responsible for communication matters, a *Pool Manager* takes care of handling several components instances, along with a number of the above subsystems. The wrapper is in the unique position of offering services to the component, which includes logging services and possibly others. Naturally, presenting a coherent view of all management aspects of components and not getting lost in complexity are conflicting goals and subject to compromise. Ideally a component's manageability would be presented in a unified view that covers the wrappers subsystem as well as the component's instrumentation. For part of this view it's possible to exploit the JVMs instrumentation to monitor performance metrics. Even though some manageability like the aforementioned performance metrics may be captured generically for all components, there will always remain aspects of a component that are specific to it and require custom instrumentation. Given the recent inclusion of JMX[8] in the JDK it is not unreasonable to assume that an instrumented component exposes its manageability through JMX. If this is not the case it should be enhanced with metadata, which for instance could be a ModelMBeanInfo object or code annotations, from which an appropriate instrumentation can be created. Both approaches should be supported to serve JMX-unaware components as well as ones that already expose management operations and attributes.

4.3 Remoting

Although MBeans may act as facades to distributed components, it will become necessary to think about *federations* of agents at some point. With a *single-agent-view* approach it is possible to hide the complexity of the federation from the management application, in other words there is a single point of access that standalone managers communicate with or where a web client is available. This is achieved by propagating requests within the federation via proxies, broadcasts, directories or some other mechanism. Suns JDMK contains a cascading service that provides such functionality and JSR255 currently attempts to standardize *agent federations* and provide a reference implementation in time for the release of JDK6.0, codenamed Mustang. From a management point of view a WSMX instance will consist of a set of WSMX kernels, organized in federations, one kernel per machine, each of which may host a set of components. This strategy allows taking advantage of locality while keeping remoteness transparent.

5 Components Collaboration

In order to enable data exchange among components a Tuple Space is utilized. It enables communication between distributed units running on remote machines. We must emphasise that components themselves are completely unaware of this distribution. That is, an additional layer of wrappers provides them with a mechanism of communication with other components.

5.1 Space-based communication

Tuple Space is shared distributed space where applications can publish and subscribe to tuples. Subscription is specified by utilizing templates and matching them against tuples available in a space. Current WSMX implementation is based on a variant of a Tuple Space, namely JavaSpaces[2]. Issues like data transfer, synchronization, persistence, etc., are handled by JavaSpaces itself, thus programmers can focus on their application objectives.

Additional requirements born out of latency need to be considered whist utilizing a Tuple Space communication. Tuple Space is composed of many distributed Tuple Space repositories that need to be synchronized. In order to maximize usage of WSMX components available within a local machine a caching mechanism should be used. Preferably, instances of distributed Tuple Space should be running on each WSMX machine and newly produced entries should be published in there. Before synchronization with other distributed Tuple Spaces takes place, a set of local template rules needs to be executed in order to check if there are any local components subscribed to the newly published event type. That is, if not otherwise specified, local components should have priority in receiving locally published entries.

5.2 Wrappers

Wrappers are generic units that separate components implementation from communication issues. *Wrappers* provide services to a component and enable communication with other components. *Wrappers* are automatically attached to each component implementation during instantiation carried out by a WSMX Kernel. There are two major parts of each *Wrapper*:

- ***Reviver*** Its responsibility is to interact with the transport layer (i.e. a Tuple Space). Through the transport layer, *Reviver* subscribes to a proper event-type template. Similarly, *Reviver* publishes result events in a a Tuple Space. Again, this level of abstraction reduces the changes required in code if the transport layer changes.
- ***Proxy*** To enable a component to request another component functionality *Proxy* is utilized. The component currently being executed might need to invoke other component functionality via *Proxy* by specifying a component name, a method to be invoked and parameters. Proxy creates proper event for this data and publishes it in a Tuple Space. A unique identifier is assigned to the outgoing event and is preserved in the response event. Proxy subscribes to the response event by specifying a unique identifier in the event template. It guaranties that only the *Proxy* that published this event will receive the result event. *Proxy* is not necessarily part of *Wrapper*, it can be also part of Dynamic Execution Semantic. This approach means that proxy calls can be redirected according to the requirements of a particular execution semantic.

5.3 Dynamic Execution Semantics

Dynamic Execution Semantic enables composition of loosely-coupled WSMX components and provides a necessary execution logic (e.g. conditional branching, fault handling, etc.). As depicted on Figure 3 an instance of Dynamic Execution Semantic is part of each event published in a Tuple Space. A *Reviver* is the thread implementation that enforces the general computation strategy of a component and operates on the transport as well as the component interfaces. It takes appropriate events from the a Tuple Space and allows execution of the attached instance of Dynamic Execution Semantic. Local component functionality can be invoked by the attached instance of Dynamic Execution Semantic, which has a state that changes over time whilst travelling and executing across distributed component locations. Additional data obtained during execution can be preserved in Dynamic Execution Semantic instance.

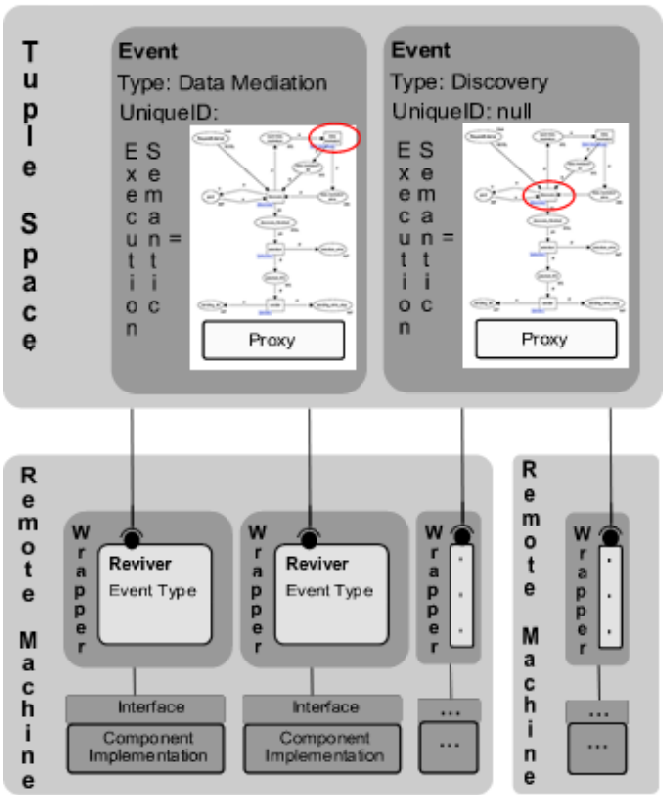

Fig. 3. Internal WSMX communication

We consider two major approaches to Execution Semantic representation in WSMX, namely as a Java code and as a workflow model. In the current version of WSMX the first approach is already fully implemented and any Dynamic Execution Semantic specified in Java can be carried out by WSMX. The latter approach represents our future vision of WSMX Dynamic Execution Semantic embedded in a distributed workflow model.

Java Dynamic Execution Semantic is represented by a state-aware piece of code that is executed whilst being fetched by a *Reviver*. The execution path is represented similarly as in Abstract State Machines and the current state is encoded in an execution instance. When Execution Semantic is executed, component implementation represented by Wrapper is exploited and additional steps can be conducted. According to the result returned from the component, the next step can be taken. The next event type is passed on to the *Reviver* and the state of Execution Semantic instance is changed. Additionally, some data can be stored for further processing in an instance of Dynamic Execution Semantic. Finally the updated event is published in a Tuple Space. Although the different parts of the execution semantics are executed on different distributed components, the Execution Semantics can be specified centrally.

We envisage workflow Dynamic Execution Semantic as a next step in WSMX component composition and coordination. General rules are similar to a case of Java code representation. The workflow approach possesses certain advantages over Java representation, but there also are some challenges. Among the advantages of the workflow approach is its graphic representation, capability to perform prior model correctness checking, flexible response to changes and on-the-fly execution of the created model. Models created for WSMX components composition should not be affected by WSMX Tuple Space communication paradigm and capability to use all available expressions (patterns) for the chosen workflow language should be preserved. A crucial aspect is to provide an instance synchronisation mechanism for executed models. It is especially relevant in cases of parallel execution where race condition might occur, thus it is necessary to ensure validity of context when executing tasks in a component. Before a task is finalized one needs to check whether context relevant for the task has changed. If the context has changed (i.e. input data and variables required for task execution) tasks must be executed again. It needs to be stressed that all aspects related to distributed workflow execution have to be considered in this case.

6 Related Work

We can distinguish two major areas in which similar works has been carried out. Distributed Dynamic Execution Semantic is related to Agent Systems, since it allows execution of state-aware code on various locations across the Internet and interaction between components is not hard-wired. On the other hand, WSMX as a Semantic Web Services system can be related to other efforts in this area. In this section these two fields of related work will be presented.

6.1 Multiagent System - JADE

JADE[1] (Java Agent DEvelopment Framework) is a Java implementation of multiagent system (MAS). It facilitates the implementation of multiagent systems through a middle-ware that provides all necessary services to enable communication compliant with FIPA specifications and set of graphical tools supporting agents debugging and deployment. FIPA[3] protocols specify communication patterns for heterogeneous and interoperable agents. JADE agents can be running within containers on various platforms on distributed machines and overall configuration can be controlled via administration framework.

Agents can move and clone themselves across distributed JADE containers. That is, agent presence within specific locations can entail additional privileges like access to local resources represented by local agents. This similarity to the WSMX Dynamic Execution Semantic approach has to be highlighted. In WSMX events include an executable part that is executed on machines that took an event from a Tuple Space. Like an agent, this executable part of an event takes advantage of local resources accessible via Wrappers.

6.2 Semantic Web Services - IRS III, OWL-S, METEOR-S

There are several other software tools providing support for execution of Semantic Web Services having their roots in OWL-S, Meteor-S and WSMO initiatives. Commercial integration platforms capable of overcoming integration problems between heterogeneous systems are also available. While none of them offers dynamic execution semantics, their functionality just on a syntactical level remains similar to functionality provided by WSMX. This section provides a short overview of these platforms.

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 [4]. 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. There is already ongoing work to achieve interoperability between WSMX and IRS III - 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 [6]. There is no comprehensive strategy for development of a Meteor-S server, as there is for WSMX or IRS III. Rather there are multiple efforts to address different aspects of Semantic Web Services. While Meteor-S tools are equal to WSMX components, it is hard to talk about any execution semantics of Meteor-S as no system comparable to that of WSMX really exists. The WSMX team plans to investigate in collaboration with Meteor-S how tools of one system could be exploited in the other.

OWL-S [3] is a comparable effort to WSMO initiative, attempting to define an ontology for Semantic Web Services. As for Meteor-S there are multiple tools

[3] http://www.fipa.org

available, but there is no integrated strategy regarding the development of a complete infrastructure for execution of OWL-S Web Services. There are some related efforts to WSMX to build an 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.

7 Conclusions

Significant improvements have been made on several fronts in the battle for a scalable, light-weight, extendible infrastructure for the execution of Semantic Web Services. A framework for component plugability has been set in place, backed by a sophisticated configuration system, archive format and classloading mechanismn that ensures the proper isolation of components. Additionally the system has been carefully engineered to support monitoring and management in a flexible, reusable fashion. By the introduction of space-based computing we referentially and temporally separate the components from each other. Wrappers in turn guarantee the transparency of remoteness and distribution from the perspective of the individual component. They also enable the central specification and distributed execution of the Dynamic Execution Semantics.

References

1. F. Bellifemine. JADE. A White Paper.
2. E. Freeman and S. Hupfer. Make room for JavaSpaces. http://www.javaworld.com/javaworld/jw-11-1999/jw-11-jiniology.html,.
3. D. Martin. OWL-S: Semantic Markup for Web Services. Technical report, 2004. http://www.daml.org/services/owl-s/1.0/owl-s.html.
4. E. Motto, J. Domingue, L. Cabral, and M. Gaspari. IRS-II: A Framework and Infrastructure for Semantic Web Services. In *Proceedings of the second International Semantic Web Conference Sanibal Island, FL, USA*, pages 306–318, 2003.
5. OASIS. Web Services Distributed Management (WSDM). http://www.oasis-open.org/committees/workgroup.php?wg_abbrev=wsdm.
6. 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*, pages 553–562, 2004.
7. D. Roman, H. Lausen, and U. Keller. Web Service Modeling Ontology (WSMO). WSMO Working Draft, 2004.
8. Sun Microsystems. Java Management Extensions (JMX) Specification 2.0. http://www.jcp.org/en/jsr/detail?id=255.
9. Maciej Zaremba and Eyal Oren. WSMX Execution Semantics. http://www.wsmo.org/TR/d13/d13.2, 2005.