Semantic Service Bus: Architecture and Implementation of a Next Generation Middleware

Dimka Karastoyanova, Branimir Wetzstein, Tammo van Lessen, Daniel Wutke, Joerg Nitzsche, Frank Leymann

Institute of Architecture of Application Systems, University of Stuttgart
Universtitaetsstrasse 38, 70569 Stuttgart, Germany
E-mail: {firstname}.{lastname}@iaas.uni-stuttgart.de

Abstract

In this paper we present a middleware for the Service Oriented Architecture, called the Semantic Service Bus. It is an advanced middleware possessing enhanced features, as compared to the conventional service buses. It is distinguished by the fact that it uses semantic description of service capabilities, and requirements towards services to enable more elaborate service discovery, selection, routing, composition and data mediation. The contributions of the paper are the conceptual architecture of the Semantic Service Bus and a prototypical implementation supporting different semantic Web service technologies (OWL-S and WSMO) and conventional Web services. Since mission critical application scenarios (for SOA) involve complex orchestrations of services, we have chosen to utilize semantically annotated service orchestrations as the applications to employ this middleware.

1. Introduction

The Service Oriented Architecture (SOA) is an approach to integrating enterprise applications in a flexible and loosely coupled manner. SOA is built on the notion of services, which are realizations of self-contained business functions and provide a service requester with an abstract view on the business functions [7]. Services can conceptually be divided into two distinct parts. Service interfaces (or service types) describe the functionality the service provides, while service implementations provide concrete realizations of these interfaces on concrete ports. This way multiple service implementations and deployments are hidden from the client behind the same abstract service interface. To provide a service requester with additional information for selecting a service implementation that is most appropriate for its needs, service interfaces can be annotated with policies to capture non-functional properties and quality of services (QoS), like reliability, security, etc. Furthermore an important aspect of SOA is service composition (orchestration). By using the two-level-programming paradigm [13], services, i.e. business functions, can be composed to form new applications, which then again can be exposed as services to facilitate further composition.

On a technological level, the Web Service technology [7, 2] is the most prominent realization of SOA today. Web Services define a set of specifications that enable standard-based service description, discovery, invocation, and composition through the use of WSDL, UDDI, SOAP, BPEL and others.

The middleware infrastructure that supports an SOA is the Enterprise Service Bus (ESB). In general, a service bus [6] enables standard-based, loosely-coupled communication among distributed services that are connected to the bus via abstract endpoints (implemented as e.g. adapters). The core functionality of an ESB can be described by the following characteristics.

By treating services as abstract endpoints, a service bus facilitates *service virtualization* [14]. This means that services are described in a unified manner and the implementations of the same service description are interchangeable. Service providers publish interface descriptions of the services they provide via the bus. The interaction of the service requesters with the services is governed by the service interface descriptions. The discovery and selection of an appropriate service implementation is delegated to the bus which uses the service description and the requester's requirements (e.g. specified through policies).

Services interact in a *loosely coupled* manner through the service bus, which has two implications on the functionality the bus has to provide. As all interactions between services are conducted through the bus, i.e. services do not have to know the concrete partner they are interacting with, the bus has to provide functionality for *routing* of messages between communication partners [6]. Furthermore, as services typically do not share a common data format, flexible means for *data transformation* have to be provided.

As service composition is an essential aspect of SOA, providing *service orchestration* is an important requirement for an ESB. Typically orchestration is realized by means of orchestration services, which are on the one hand service requesters, as they interact with partner services over the ESB, while on the other hand they provide the service composition as a service to other service requesters through the ESB. The Business Process Execution Language for Web Services (BPEL) [3] is the de-facto standard for describing workflow-like compositions of Web Services (i.e. Web Service orchestration). In BPEL composite services are built by combining activities that represent interaction with Web Services (invoke, receive, reply, pick) with control flow activities (flow, sequence, while). Web Service partners of BPEL processes are specified using a two-level approach. On an abstract level the service type (WSDL portType), i.e. the interface supported by the partner service, is defined. This abstract partner description is complemented by design-time or deployment-time information defining a concrete Web Service implementation (WSDL port).

In traditional ESBs service *discovery* is limited to finding implementations of service interfaces. Assuming usage of Web service technologies, the service interfaces are described in WSDL, which only defines syntax, but not the semantics of a service functionality. WSDL describes a service interface in terms of port types and operations. WSDL operations can define parameters which are typically defined in XML Schema. When a requester hands over a service type described in WSDL to the service bus, the bus can only find services that implement exactly the same service interface. If another service provider implements the same functionality but with a different signature, the service would not be considered by the bus as a compliant one. Another important aspect are mismatches on data types. The data the service requester and provider expect are often of different types. For the invocation to take place, the bus has to transform or *mediate* between the two data formats. In traditional ESB technology such a transformation is specified at design-time and executed at runtime, e.g. as an XSL-transformation.

Technologies emerging from the field of Semantic Web Services (SWS) offer new opportunities to enrich the functionality of an ESB. When using SWS the service semantics is explicitly defined, and can be exploited by the bus to find services, potentially offering syntactically different interfaces, thus increasing the pool of alternative services. When using SWS technologies automatic run time data transformations can be done based on the semantic information available in the semantic descriptions of web services.

We identify the need for an enhanced middleware to facilitate the use of semantics in an SOA environment. We call such a middleware a Semantic Service Bus (SSB). The contributions of this paper are the architecture of the SSB, and a prototypical implementation that supports two SWS technologies (OWL-S and WSMO) and WSs.

The paper is organized as follows. In the next section we stress on the advanced features of the SSB distinguishing it from conventional service bus infrastructures. In section 3 we present the architecture of the SSB. To show the feasibility of using the features an SSB must provide, we use semantically annotated processes as an application scenario. For this we shortly discuss approaches for enriching existing service orchestrations with semantic information (section 4). Our first experimental implementation of a Semantic Service Bus is presented in section 5. Finally, we give conclusions and directions for future work.

2. Semantic Service Bus

The Semantic Service Bus introduces a new level of *abstraction*. While the conventional service bus is restricted to using interface descriptions of services, in terms of messages they consume and produce, the Semantic Service Bus requires as input only functional requirements to services regardless of their interface description. One approach to describe the functional requirements to services that has gained broad acceptance is the use of semantic descriptions [16]. Applying this approach to the Web Service technology yielded the Semantic Web Services. Prominent existing examples of this approach are OWL-S [15] and WSMO [8], briefly described later in the paper. When using the semantic descriptions of Web Services, we advertise their functionality rather than their signature. Hence, we do not restrict the use of a service that implements a particular WSDL description, but allow the use of any Web Service implementing the required functionality regardless of its interface description.

Similarly to the conventional service bus, the Semantic Service Bus is the middleware used by applications to invoke services, too. It has to deal with service discovery, message routing and message transformation, which in addition implies using the semantic knowledge about these services.

The Semantic Service Bus takes as input from applications (i) the requirements to services represented semantically in order to perform semantic service discovery and (ii) the data the service needs to perform its functionality (and QoS). A set of compliant SWSs is discovered, and a single service is selected by the bus based on the criteria specified by the service requesters. The bus invokes the selected service using its binding information and sends the response of the invocation back to the requester. The SSB must support both synchronous and asynchronous modes of communication between the service requestors and service providers. Most importantly, it must enable interaction with stateful services. The existing Semantic Web Service technologies however, still address this issue inadequately. Theoretical foundations for enabling the interaction with stateful services are made available. However the interaction of service requesters and service providers is restricted to rigid client-server communication.

It is well known that data models even within a company are different, i.e. they may be standardized within a single department, but are usually not standardized company-wide, let alone between several companies. This requires mapping between data models. Using the conventional service bus one can only hard-code data transformations, e.g. using a BPEL process as mediator, or an XSL-transformation. The Semantic Service Bus allows the specification of *data mediation* (data transformation) in an abstract manner. This is achieved by using the concept of mediation, i.e. the data types are enriched with semantic descriptions thus enabling the transformation of data types using the semantic knowledge (including reasoning). Introducing this new level of abstraction - the *virtualization of data transformation* - any service that fulfils the required transformation functionality can be used, be it a mediator, XSL-transformation, BPEL process or special purpose application.

The Semantic Service Bus exhibits *improved routing* procedures, too. Semantically enriched routing itineraries can be defined, in the form of BPEL processes for instance [10], and executed using semantic discovery and transformation.

3. Architecture

Like a conventional service bus the Semantic Service Bus provides means that enable applications and services (i) to plug in to the bus, (ii) to make use of its infrastructure services, (iii) configure the bus and (iv) manage the available SSB components that provide services.

The semantic service container and the invocation and management framework, depicted in figure 1, are the facilities a service or application must use to plug in to the SSB. Additionally they are used by the services to communicate with the SSB. The service container and the invocation and management framework in combination exhibit analogous functionality as in an ESB [6]. In both cases they are the physical manifestation of abstract endpoints. These two components enable the virtualization of applications with respect to their implementations, technology used, the transport protocol they require and their mode of communication (synchronous or asynchronous).

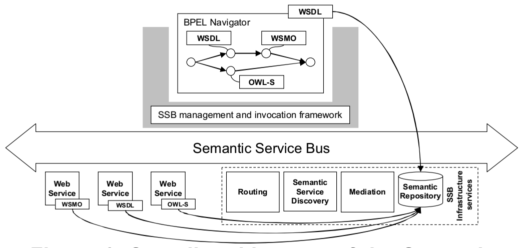


Figure 1. Overall architecture of the Semantic Service Bus.

The Semantic Service Bus, like any other middleware, provides also a set of *infrastructure services*. These include service registry, discovery services, transformation or mediation services, orchestration, routing, security, logging and others. These are inseparable parts of a Semantic Service Bus and implement the advanced features explained in section 2. For the sake of simplicity only those infrastructure services relevant to the enhanced features of the bus are shown in figure 1. Apart from the infrastructure services other services can be provided on the bus at any time, e.g. WSDL, WSMO and OWL-S services (Figure 1).

The management facilities of the bus include the configuration console and the management interface. The configuration console enables manipulation of the setup of the bus. Configuring the bus incorporates selecting components that provide infrastructure services to the bus and components that implement technology-specific service containers. The management interface offers external access to the management capabilities of the services plugged to the bus (Figure 2).

The applications using this SOA middleware need to plug in to the bus and supply it with their requirements to the services they want to invoke. These requirements may be expressed in terms of WSDL port types, or in a declarative manner using any Semantic Web Service technology, e.g. WSMO and OWL-S. In addition other standards like WS-Policy may be used to state requirements on the QoS characteristics of services. Applications that use the bus to expose services (like service modelling tools) must use the management interface to deploy them. This is illustrated by the deployment scenario shown in figure 2. The deployment of services on the bus starts with the deployment of a service into the service container (for example a BPEL engine) using its deployment facilities via the bus's management interface. Then the management facilities of the SSB infrastructure services are used to register the service at the (semantic) service repository. After this deployment procedure the service is made available via the bus.

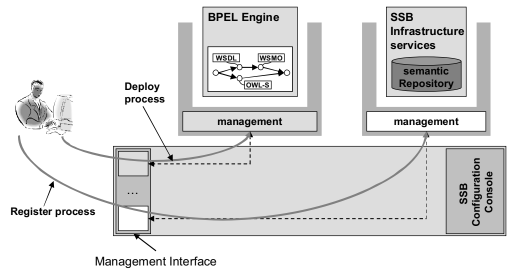


Figure 2. Management capabilities of the Semantic Service Bus.

A prototypical implementation of a simple Semantic Service Bus is presented in section 5. Before this, in the next section we give an overview of semantically enriched processes, since these are the use case scenario we employ for manifesting the usage of the SSB.

4. Semantically Annotated Processes

Seamlessly orchestrating services is one of the prominent features of SOA. Being able to orchestrate services gives huge advantages to companies, because they can compose existing and new services, company-grown and external applications likewise. In SOA service compositions are also exposed as services and can be used like any other service. On the other hand a service orchestration needs a middleware to support the discovery and invocation of the involved service implementations. Therefore, service compositions are one example of applications that need to plug in to the bus to be able to use its capabilities.

Generally, service orchestrations (e.g. BPEL processes) are executed by a process execution environment, also called process engine. A BPEL engine is hence one component on the bus that provides a composition service to the bus, and on the other hand needs to utilize the Semantic Service Bus for the execution of services.

If one uses vanilla Web Services the SSB must support the Web Service technology. Apart from the port type and/or operation definition no additional information is needed to invoke a Web Service using the Semantic Service Bus. In order to interact with Semantic Web Services the activities in a BPEL process need to be semantically annotated to express the requirements to services. Multiple semantic WS technologies can be used in combination to model/define the semantic information associated with a single process definition. This entails that the SSB must support these SWS technologies.

There are several different approaches to annotate process activities with semantics; they are orthogonal to the technology used to describe the semantic information:

1. WS-Policy Framework:
   The WS-Policy Framework [5] supports two ways to assign semantic descriptions to a BPEL activity. The first one is by directly referencing a policy from within an activity, the second one - by attaching policies to existing process activities using WS-Policy attachments.
2. Extensions to the BPEL language:
   BPEL is an extensible language and its extension mechanisms can be used to define new language elements. These can be used to point to relevant semantic information or to inline such information into the BPEL processes themselves.
3. Use of Deployment Information:
   The BPEL specification postulates that the resolution of concrete endpoints is a deployment-related issue [3]. For this reason, standardizing the deployment information is out of the specification's scope. In order to execute BPEL processes deployment information must therefore be provided in the format required by the particular engine, e.g. using a deployment descriptor. Requirements to the capabilities of the services involved in a process can also be a part of the deployment information, which are then used as input to the Semantic Service Bus the engine utilizes.

In the Semantic Service Bus prototype presented in the next section the semantically annotated processes have been used as a scenario to showcase the use and the viability of the Semantic Service Bus. Any other service requesters (different from services) can also make use of the SSB.

5. Implementation

As mentioned in the previous section a BPEL engine is one kind of application that can be plugged in to the Semantic Service Bus both to use the SSB as a middleware and to provide infrastructure service implementations to the SSB.

The SSB infrastructure services may have multiple implementations. One main reason is that a Semantic Service Bus must support both Web Services and Semantic Web Services. In addition, there are different Semantic Web Service technologies.

The engine we use in this work is the ActiveBPEL engine [1]. This engine comes with an Axis[1] playing the role of both service container and a management and invocation framework of a service bus suitable for using WSs described in WSDL. We have extended the engine

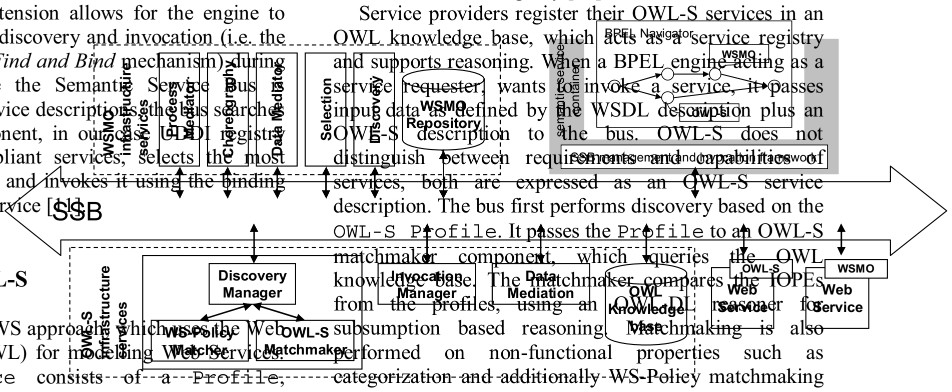
[1] http://ws.apache.org/axis/

component that connects to the service container to enable it to interface the SSB so that it can use the other infrastructure service implementations as well. We have experimented with the OWL-S and WSMO technologies (sections 5.2 and 5.3), as well as Web Services described in WSDL (section 5.1). Therefore some of the infrastructure services have been implemented from scratch, while others have been implemented by reusing existing Semantic Web Service frameworks (e.g. IRS-III [9] and WSMX [18]).

The prototype we present supports service discovery based on WSDL or semantic descriptions of services and service invocation. In fact the SSB supports in this way dynamic binding to services. For this the so-called *Find and Bind* mechanism [2] is supported. The mechanism boils down to performing (i) a lookup *(find)* of service implementations (i.e. ports) based on the service description (ii) *binding* and (iii) *invocation* of the discovered service. For each particular semantic Web service technology and service description language the discovery, selection and invocation of services the SSB has to support are implemented differently.

5.1 Support for dynamic binding to WSDL services

To facilitate discovery of service ports based on WSDL service interface descriptions, the invocation and management framework of the service container in which the ActiveBPEL engine runs has been extended. The functionality of this extension allows for the engine to utilize the bus both for discovery and invocation (i.e. the implementation of the *Find and Bind* mechanism) during process runtime. Once the Semantic Service Bus is provided the WSDL service descriptions, the bus searches a WS discovery component, in our case UDDI registry (jUDDI [4]), for compliant services, selects the most appropriate service port and invokes it using the binding information about the service [11].



5.2 Support for OWL-S

OWL-S [15] is an SWS approach which uses the Web Ontology language (OWL) for modeling Web Services. An OWL-S `Service` consists of a `Profile`, `Process` and a `Grounding`.

Profile defines "what" the service does and is used for Service discovery. It contains a description of inputs, outputs, preconditions and effects (IOPEs) of the service, non-functional properties such as service categorization (e.g. NAICS, UNSPSC), and additional user-defined service parameters. The inputs and outputs are specified by pointing to concepts from domain ontologies.

Process defines "how" to access the service, i.e. in which order and under which conditions the operations of the service are to be invoked.

Grounding defines a mapping of the `Process` to WSDL operations. The mapping from OWL-S operation parameters, which are specified as OWL types, to WSDL types, which are typically defined in XML Schema, is done by XSL-transformations.

Through the use of OWL-S the Web Service interface is given explicit semantics, enabling the service bus to perform discovery and invocation of semantically identical services automatically, even if the syntax in the WSDL descriptions is different. The `Profile` is used for discovery while `Process` and `Grounding` are used for data mediation and invocation of the service. Note that the WSDL binding is used for the concrete invocation of the service.

Our SSB prototype, as shown in figure 3, supports OWL-S technology through the use of OWL-S specific infrastructure services including discovery, invocation, data mediation and registry [11].

Service providers register their OWL-S services in an OWL knowledge base, which acts as a service registry and supports reasoning. When a BPEL engine acting as a service requester, wants to invoke a service, it passes input data as defined by the WSDL description plus an OWL-S description to the bus. OWL-S does not distinguish between requirements and capabilities of services, both are expressed as an OWL-S service description. The bus first performs discovery based on the `OWL-S Profile`. It passes the `Profile` to an OWL-S matchmaker component, which queries the OWL knowledge base. The matchmaker compares the IOPEs from the profiles, using an OWL-DL reasoner for subsumption based reasoning. Matchmaking is also performed on non-functional properties such as categorization and additionally WS-Policy matchmaking takes place. The discovery service returns a list of compliant services, from which one service is selected for invocation.

Figure 3. Architecture of the Semantic Service Bus prototype.

The selected service has the same semantics as the requested service, yet it could have a different WSDL interface, especially different data type definitions. Thus, it is necessary to lift the input data as provided by the requester to its ontological OWL representation as defined in the OWL-S description and then lower it to the XML as required by the service provider. Only in this way can the selected service be invoked. This lifting and lowering, also called data mediation, is provided by XSL-transformations which are specified in the OWL-S grounding. The WSDL service can finally be invoked as specified in the corresponding WSDL binding. After invocation of the service and another round of lifting and lowering the output data is sent back to the requester.

Our implementation is based on the OWL-S API [17] that provides an OWL knowledge base and supports execution of an `OWL-S Process` and invocation of a WSDL Web Service using the `Grounding`. We have implemented the OWL-S matchmaker, the data mediator and the WS-Policy matchmaker [11].

OWL-S supports the description of stateful services through the use of the process model (OWL-S `Process`).

The service requester uses the process model to communicate with the service in a conversation, which can potentially span several operations.

One shortcoming of the OWL-S process model is that it doesn't support callback operations, i.e. operations the service should invoke on the service requester.

Another problem arises when a stateful service is to be discovered and invoked dynamically by the bus. In this case the discovery has to be extended to also include matchmaking of process models of the service requester and the service provider. This is a difficult task and actually a process mediation problem; it is part of our future work.

5.3 Support for WSMO

The Web Service Modelling Ontology (WSMO) is a competing technology to OWL-S for semantic description of Web Services. It relies on four major modelling elements [8]:

Ontologies provide the terminology and formal semantics used by other WSMO elements in order to describe relevant aspects of application domains. Therefore ontologies define concepts, relationships between concepts and rules to build a valid and unified domain terminology.

Web Services wrap real world Web Services and enrich them by describing their functional, behavioural and non-functional properties semantically. Therefore WSMO introduces the concepts of `Capabilities` and `Interfaces`. Capabilities describe the functional aspects of the offered services in terms of pre- and post-conditions, assumptions and effects. While pre- and post-conditions describe the state of the local information space (i.e. input/output data) before and after the service invocation, respectively, assumptions and effects refer to state and its modifications in a global point of view. Interfaces give information about the operations of the service in terms of choreography and orchestration. The choreography defines the behavioural interface that service consumers need to comply with in order to communicate correctly. Herewith WSMO enables dynamic process mediation on a conceptual level. The orchestration on the other hand provides details on how the service works (from the provider's point of view) to achieve its capabilities.

Goals describe the user's need of a Web Service solving a specific problem. Therefore, the goal defines the preconditions and assumptions as well as effects and post-conditions the desired service should provide. Using this functional description, a reasoner can discover WSMO Web Services, which provide the requested functionality.

Mediators provide concepts to define links between the other WSMO modelling elements in order to resolve heterogeneity problems. Mediators are useful to define links between `Goals` and `Web Services` (using `wgMediators`) or merge terms from different domain ontologies (using `ooMediators`). To reuse existing goals or to refine abstract goals, `ggMediators` can be used while `wwMediators` enable the modeller to define relations between Web Services, e.g. to state that one Web Service rolls back another one [12].

The explicit distinction between `Web Services` and `Goals` WSMO makes, is one of its main differences to OWL-S.

Furthermore, WSMO introduces a predefined set of non-functional properties which can be used to attach metadata to modelling elements. When necessary, this set of properties can be extended (e.g. to add QoS requirements) [12].

On a conceptual level WSMO enables invocation of stateful semantic Web services. However this is limited to a rigid client-server interaction, where the WSMO Web service offers some 'operations' that lead to a goal fulfillment. But it is not possible to specify that the client that wants to achieve a goal offers a callback operation that the called service has to use in order to report the result back to the client.

Currently, two WSMO execution environments exist. The Web Service Model eXecution environment (WSMX) is regarded as the reference implementation of WSMO.

It specifies four entry points [18] to enable clients to discover and/or invoke WSMO-enabled Semantic Web Services fulfilling a specific `Goal`. The other implementation, the lisp-based Internet Reasoning Service (IRS-III) [9], uses an extended WSMO to perform capability-based Web Service invocation by means of `Goals`. Both are available frameworks for WSMO, in different development stages.

In the context of this work, we have implemented a management and invocation framework extension at the BPEL engine's service container that delegates the invocation of services required by semantically annotated BPEL invocation activities to both WSMO implementations using the entry point `achieveGoal(WSMLDocument):Context`. Both frameworks (WSMX and IRS-III) are called synchronously, with WSMO goals submitted to their entry point APIs, which take care about discovering and invoking an appropriate service and sending the response back to the requester. The WSMX discovery is done by comparing the requested capabilities of the goal with the capabilities of the semantic Web services available in the repository, whereas IRS-III performs discovery by means of WG-mediators that indicate that a goal can be fulfilled using a certain Web service.

In general, a goal is not restricted to synchronous invocation; it may contain a complex choreography description. If this is the case, WSMX tries to discover a service with a choreography description that represents the counterpart to the goal's choreography; IRS-III does not support such functionality. Then there may be a longer communication between the SSB and the discovered (stateful) service. When the result is computed it is sent back to the requester.

Conceptually WSMX even supports asynchronous invocation of stateful semantic Web services including client interaction via the entry point `invokeWebService(WSMLDocument, Context):Context`. In this case the longer communication would not only take place between WSMX and the discovered and selected service. Messages could also be sent from the service (via WSMX) to the client as intermediate results and the client may send intermediate messages to the service (via WSMX). However this is not yet implemented.

6. Conclusions

In this paper we have presented the concept of Semantic Service Bus and its conceptual architecture. A major characteristic of the Semantic Service Bus is the virtualization of services rendering them independent of not only implementation specifics, but also of any service interface description language (syntax) and naming conventions. The SSB uses semantic descriptions of services to specify service capabilities as well as functional, non-functional and QoS requirements to services. The semantics of services is used to enable enhanced features like semantic service discovery, semantically enhanced itineraries for routing, and mediation. These features additionally distinguish the Semantic Service Bus from conventional SOA middleware.

Due to the existence of several competing Semantic Web Service technologies (annotating Web Services with semantics) the SSB has to provide support for multiple semantic description languages. In a Web Service environment, the SSB must support WSDL service descriptions, too. This also entails support for multiple kinds of discovery components (e.g. UDDI, OWL knowledge bases, etc.). An SSB must provide entry points for applications that need to use the services it provides. It must also provide a management facility to support configuration and management of the component comprising the bus.

We presented a prototypical implementation of a Semantic Service Bus that supports the OWL-S and WSMO technologies. The prototype is also capable of discovering and invoking WSDL Web Services.

So far it is not possible for a client to communicate with the SSB asynchronously when the WSMO and OWL-S infrastructure services are used. This is mainly due to the fact that the existing framework implementations support only synchronous interaction with the middleware. This hampers the interaction of applications with stateful services, where the explicit control of the applications over the communication is required. In WSMX, to the best of our knowledge, asynchronous interaction is possible only between the bus and a stateful service; the interaction between the service requester and bus is kept synchronous. Both synchronous and asynchronous communication with WSDL Web Services via the SSB is possible. Enabling asynchronous communication between the SSB and service requesters, as well as improving the infrastructure service implementations for OWL-S and WSMO, are going to be considered in our future research work. We envision the semantically annotated BPEL processes to serve as process mediators (on behalf of the SSB) and thus enable stateful communication between service requestors and providers via the SSB. Of special interest for us is the creation of routing itineraries for the SSB based on service compositions defined in BPEL. Defining such routing procedures enriched with semantic information for Web Services and Grid Services is also part of our future work.

7. Acknowledgment

This work has been partly funded by the European Union's 6th Framework Programme, within Information Society Technologies (IST) priority under the SUPER project (http://www.ip-super.org) and the TripCom project (http://tripcom.org).

References

[1] ActiveBPEL.org. ActiveBPEL. `http://www.activebpel.org/`, 2006.
[2] G. Alonso et al. *Web Services*. Springer, 2003.
[3] T. Andrews et al. Business Process Execution Language for Web Services, Version 1.1. Specification, May 2003.
[4] Apache Software Foundation. jUDDI, Version 0.9. `http://ws.apache.org/juddi/`, 2005.
[5] D. Box et al. Web Services Policy Framework (WS-Policy). May, 2003.
[6] D. A. Chappell. *Enterprise Service Bus*. O'Reilly, 2004.
[7] F. Curbera et al. *Web Services Platform Architecture: Soap, WSDL, WS-Policy, WS-Addressing, WS-Bpel, WS-Reliable Messaging and More*. Prentice Hall PTR, 2005.
[8] J. de Bruijn et al. Web Service Modeling Ontology (WSMO). WSMO Final Draft, `http://www.wsmo.org/TR/d2/v1.2/`, April 2005.
[9] J. Domingue et al. IRS-III: A Platform and Infrastructure for Creating WSMO-based Semantic Web Services. In *WIW 2004, WSMO Implementation Workshop 2004*, 2004.
[10] F. Juchart. Development of a Routing Procedure for SOAP Messages Based on BPEL. Diploma Thesis. `http://www. informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2460`, 2006.
[11] D. Karastoyanova, F. Leymann, J. Nitzsche, B. Wetzstein, and D. Wutke. Parameterized BPEL Processes: Concepts and Implementation. In *Proceedings of BPM*, 2006.
[12] R. Lara et al. A Conceptual Comparison of WSMO and OWL-S. *Proceedings of ECOWS 2004*, 2004.
[13] F. Leymann. Web Services: Distributed Applications without Limits. *Business, Technology and Web*, Leipzig, 2003.
[14] F. Leymann. The (Service) Bus: Services Penetrate Everyday Life. In ICSOC, volume 3826 of *LNCS*. Springer-Verlag Berlin Heidelberg, December 2005.
[15] D. Martin et al. OWL-S: Semantic Markup for Web Services. W3C Member Submission, W3C, 2004.
[16] S. A. McIlraith, T. C. Son, and H. Zeng. Semantic Web Services. *IEEE Intelligent Systems*, 2003.
[17] MINDSWAP Group. OWL-S API. `http://www.mindswap.org/2004/owl-s/api/`
[18] M. Zaremba. WSMX Execution Semantics. WSMXWorking Draft D13.2 v0.3, 2005.