An ESB implementation for airport environment

P. FOGLIA, C. A. PRETE, R. SODINI, M. SOLINAS
Dipartimento di Ingegneria dell'Informazione
Università di Pisa
Via Diotisalvi, 2 - 56122 Pisa
ITALY
{foglia, prete, rita.sodini, marco.solinas}@iet.unipi.it

Abstract: - Today companies have to deal with the problem of improving their services by providing new functionalities, due to customer requests, marketplace needs and rapidly changing technologies. Old systems can't be fully replaced because they represent a reliable resource on which companies are still basing their core-business, so there is a need to extend companies' functionalities without changing all or some of their applications. In an airport environment, this is also true because airport systems need to communicate among themselves with little or no change. An Enterprise Service Bus (ESB) is an emerging software architecture that provides a communication and integration paradigm, in which a set of heterogeneous applications become able to communicate among themselves. In this paper, we discuss the architecture of our ESB implementation, which is suited for application integration in the airport environment.

Key-Words: - Enterprise Service Bus, ESB, SSB, Legacy Systems, Application Integration, J2EE, JMS, XML, XSLT.

1 Introduction

One of the main problems that we have to deal with in a business context is let new and old applications communicate among themselves [11]. This need comes from both customer and competitive pressure [6].

In this context, the first requirement that must be satisfied is to overcome the heterogeneity of existing applications. Such applications could be custom-built, acquired from third-party, part of a legacy system, or a combination of thereof, and could operate on different platforms [10]. The management of such heterogeneity is a very strict requirement, because in critical business-oriented application environment legacy applications can't be simply left with the purpose of replacing them with new systems based on more actual technologies. Legacy systems are still providing most of the business functionalities, so replacing it would mean giving up a tested system to adopt an untested one, and also readapting old data to the new format.

An *Enterprise Service Bus* (ESB) is an emerging software architecture that provides communication and integration functionalities. An ESB is generally designed to let new and old applications exchange messages with the purpose of extending functionalities of each application. Implementations of ESB must be based on open standards, because such standards reduce purchase and implementation costs, and also unlock users form vendor's proprietary integration framework [1,9].

A typical context in which this need of integration is shown is the airport environment. In this context, there are a lot of information systems that need to communicate either among themselves or with information systems of different airport areas. In such environment, the communication is actually based on proprietary paradigms. Nevertheless, the growing need of integrating new applications in such systems makes this airport environment a candidate to adopt an integration solution based on ESB. However, the standard ESB solutions can't be adapted easly to this environment, because airport systems are legacy systems based on closed architecture; for this, the main effort in adopting a commercial solution regards the development of connector technologies, which is a resource consuming task.

Our idea, that comes from typical requirements of this environment, aims to develop an ESB that:

1. provides a communication paradigm for typical airport systems;
2. minimizes the overhead that comes from connector technologies design and development.

In this paper we describe our implementation of ESB technology, developed to integrate applications operating in the airport environment. In particular, we focus on our ESB central component. This component, called *Enterprise Manager* (EM), provides typical functionalities of an ESB such as message validation and transformation, content-based routing, access control.

This paper is organized as follows: section 2 is about ESB characteristics, but also briefly describes some commercial ESB implementations; section 3 describes the architecture of our ESB implementation, focusing on its central layer; in section 4 we show two examples that have been built to better understand each component's role and how it works; section 5 contains conclusions and present a possible development of our solution.

2 ESB characteristics and implementations

An ESB is a "*standard-based integration platform that combines messaging, web services, data transformation, and intelligent routing to reliably connect and coordinate the interaction of significant numbers of diverse applications*" [1].

The ESB architecture has been developed to overcome the limits of proprietary integration solutions [1,9]: it allows an highly-distributed integration, letting a set of

individual applications work together basing on a communication paradigm, and also letting them be scaled independently from one to another. This integration is standard-based (for example J2EE [7] components are widely adopted) otherwise ESB fall again in the limits of proprietary solutions. Moreover, an ESB must provide an event-driven, document-oriented processing model based on XML [12] standard, because *i*) the latter is ideal for sharing structured data among heterogeneous applications; *ii*) document's content transformation (one of the most important characteristics in a heterogeneous applications integration) becomes very easy. And ESB also provides support for message routing based on messages content, so that producers send messages to a single channel, and the routing component forwards it to the correct destination, avoiding coupling among applications [10]; in this way it is possible to guarantee the integrability of new systems and their modification with no repercussions on the others.

Many IT companies have developed an integration platform based on ESB technology. Here we present a brief summary of the characteristics of some of these solutions.

The Fiorano ESB [2] supports intelligently-directed communication and mediated relations among business components. It also supports both SOA [16] and EDA [17] over a single technology base, together with tools, such as security and administration tools.

WebSphere [3] is an integration platform developed by IBM, that provides two solutions: WebSphere ESB that provides standard-based web services integration, and WebSphere Message Broker that allows for applications that do not conform to standard for connecting to an ESB.

Sonic ESB [4] simplifies the integration of business components using a standard-base, SOA, and letting architects dynamically configure the connection, mediation and control of services and their interaction. These characteristics make it easy to deploy initial project and to scale and evolve them.

Artix [5] is an ESB designed for complex, mission-critical enterprise integrations challenge. It provides extensibility, thanks to its plug-in architecture, comprehensiveness, due to the mobile-to-mainframe platform support, and support for security.

As already stated, such commercial solutions are not suitable for the airport environment, because of the need of redesign adapters for airport information systems applications due to their non-standard interfaces.

3 Smart Service Bus

The Smart Service Bus (SSB) is the implementation of an ESB suitable for the airport environment. The architecture of SSB is divided into a set of layers, and each layer can be composed by one or more components (Fig. 1). In this way, we can demand every responsibility to the component designed to manage it, so that we can maintain each responsibility separated from other components.

Each layer implements the following functionalities:

Technology Connector Layer: this layer performs the technology integration. It is composed of *Technology Connectors* and *SSBConnector*;

Enterprise Server Layer: this layer is the heart of the SSB logic. It manages message control and flow, and also performs access control and data translation. It is composed of *Enterprise Manager* (EM), and of the validation, transformation and routing components;

Message Layer: provides the support for message exchange. It is only composed of *Enterprise Service Backbone*.

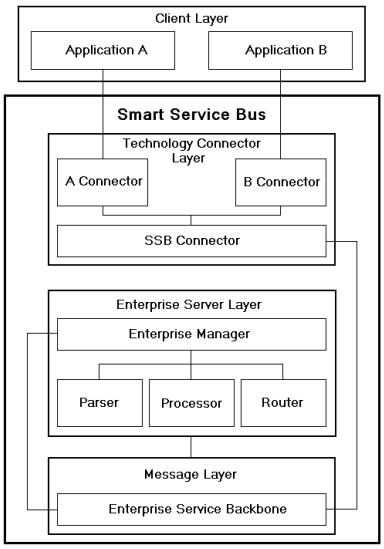

Fig. 1 SSB architecture

The *Enterprise Service Backbone* represent the communication bus of the SSB, and all types of messages travel above it.

The *Enterprise Manager* is the component that manages message validation, transformation and routing, and also performs a security access control. It does this by the coordination for three components: the *Parser*, that is responsible of the validation; the *Processor*, that has to manage the access control and the transformation; and the *Router*, that can forward a message toward the correct destination via the *Enterprise Service Backbone*.

The *Technology Connector* has to provide the interface that let each kind of system communicate via SSB. We have a technology connector for every kind of application (for example, CORBA Connector, HTTP Connector). These components are the only part of the SSB that applications can see, and each of them doesn't communicate directly with the bus, but uses the services provided by the *SSBConnector*.

3.1 SSB Middleware

The communication technology is an implementation of JMS [8], while messages are written in XML language.

The SSB structure is designed to use both types of channels that JMS offers: queue and topic [8].

Queues are used to manage client-server and administrative communication, while topics are used to manage event delivery.

As a result, in the SSB two types of communication paradigms are provided: point-to-point (p2p) [8], based on queues, and publish-subscribe [8], based on topics.

3.2 SSB Messages

In the SSB, we have three types of messages: a first type, that includes service messages, like messages to create new channels, or to let a system register itself as a service provider; a second type, that comprises messages for service invocation and for response delivery; a third type, that represents events that occur in the system.

Each category contains a set of messages that represent a specific functionality provided by the SSB, a service invocation, a response to a request, or an event.

3.2.1 Service Messages

This is the first type of message. These messages are sent by applications that need to communicate directly with the *Enterprise Manager*; we can divide them into two sub-types: the first sub-type, that includes messages sent to manage the p2p communication, for example to let a system register itself as a service provider, or to resolve a specific service; the second one, that includes messages sent to manage the events delivery, for example to let an application ask the EM for the authorization to publish or receive events.

3.2.2 p2p Messages

These messages are sent by an application that needs to make use of a specific service, and that previously has obtained the permission by EM to invoke it, by a request delivered as a *Service Message*. Moreover, the answer of the service provider is also sent as a p2p message.

3.2.3 Event Messages

This represents the last type of SSB Message. An application that has previously received the permission by EM for publishing events of a specific type, when an event of such type occurs in the publisher, it builds and sends an Event Message via SSB, that will be received by all subscribers that have previously obtained the permission by EM to receive that type of event.

3.3 Enterprise Server Layer

The heart of SSB logic stays in the Enterprise Server Layer. This logic layer of the entire architecture is split into four main components: the *Enterprise Manager*, the *Parser*, the *Processor*, and the *Router*.

3.3.1 Enterprise Manager

The main role of the EM component is to coordinate the other three components, but it also listens for specific JMS queues for message arrival. These queues are: *SERVICE_CHANNEL*, used by applications that send Service Messages; *DATA_CHANNEL*, in which p2p Messages are sent; *EVENT_CHANNEL*, from which EM receives Event Messages.

To manage all possible arrivals of messages through these different queues, the EM is split in five sub-components shown in Figure 2, three for listening on the queues, and two designed to offer administrative functions such as naming and directory services (using an LDAP [14] server), peer registration and deregistration, and so on. The first three sub-components use functionalities provided by the other two to manage the current operation triggered by a message arrival in their specific queue. These five components are:

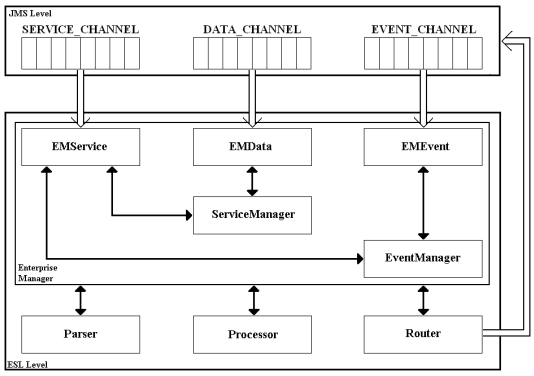

Fig. 2 EM Architecture

EMService: it is responsible for processing a request sent through the *SERVICE_CHANNEL* queue. When a Service Message is received, this component forwards it to the Parser for validation and then to the Processor for access control, and if no problems occur, EMService analyses the message to decide if it is a service request for p2p communication (peer registration or resolution, for example) or for event delivery. If this message belongs to p2p communication, EMService invokes the appropriate functionality of the ServiceManager component; if it belongs to event delivery, EMService invokes the appropriate functionality of the EventManager component.

EMData: it is responsible for processing a request sent through the *DATA_CHANNEL* queue. When a message is received, this component forwards it to the Parser for validation and then to the Processor for access control, and if no problems occur, EMData controls if it is possible to send the message, invoking the appropriate method of ServiceManager, and eventually forwards it to the Router. Once that the message has been forwarded to the right destination, EMData waits for the response message. When it arrives, the message is validated and processed and if no errors occur it is sent to the requestor by the Router.

EMEvent: it is responsible for processing a request sent through the *EVENT_CHANNEL* queue. When an Event Message is received, this component forwards it to the Parser for validation and then to the Processor for access control, and if no problems occur, EMEvent controls if it is possible to send the message, invoking the appropriate method of EventManager, and eventually

forwards it to the Router. Then it builds the response massage and sends it to the publisher to communicate it the result of the operation.

ServiceManager: this component's role is to manage operations to check if there are all the requirements to fulfil the current request. In detail, it verifies the existence of the queue that corresponds to the service that's in the current request; moreover, it carries out checks that belong to the request in LDAP, and if necessary updates it.

EventManager: this component's role is to manage operations to check if there are all the requirements to fulfil the current request. In detail, it verifies the existence of the topic that corresponds to the events that are in the current message; moreover, it carries out checks that belong to the request in LDAP, and if necessary updates it.

3.3.2 Parser

This component verifies that an XML message is valid and well-formed. The Parser is invoked every time that a message arrives to the EM, and every time that a messages is sent by the EM. If the Parser doesn't validate a message, the corresponding operation fails.

3.3.3 Processor

This component has been designed to perform access control and message transformation. To do this, it uses a set of XSLT [13] style-sheets, one for every type of SSBMessage. These style-sheets define a set of rules to be applied to the original message for its transformation. In detail, it defines the access control rules by reading the sender's identity and type of operation that the sender wants to do; if the rule decides that the sender can't access to that particular operation, the Processor refuses the request, otherwise accepts it; moreover, if required by the current operation, the Processor transforms the informative content of the message, so that the receiver is able to interpret it. This translation step is very important for the integration of heterogeneous systems, and is one of the main characteristics of ESBs [1,9,10].

3.3.4 Router

This component forwards an incoming message through the right destination, basing its forwarding rules on the message's content; it has been designed as a *Content Based Router* [1,10]. The communication paradigm could be either synchronous or asynchronous. If the message is an event or a request reply, it is sent asynchronously to the destination topic or queue; otherwise, if the message is a service invocation (p2p Message) the communication paradigm is synchronous, and when the Router receives the reply, forwards it to the EM.

4 Examples

To highlight the relationship among EM's subcomponents, in the following we give two simples examples about a set of operations that can be done with our SSB. The first one refers to services (see Fig. 3), while the second to event delivery (see Fig. 4-5).

4.1 Providing service S

Let's consider two applications that can communicate via SSB, and let's call them A and B. Let's assume that A registers itself as a provider for service S, while B wants to invoke this service.

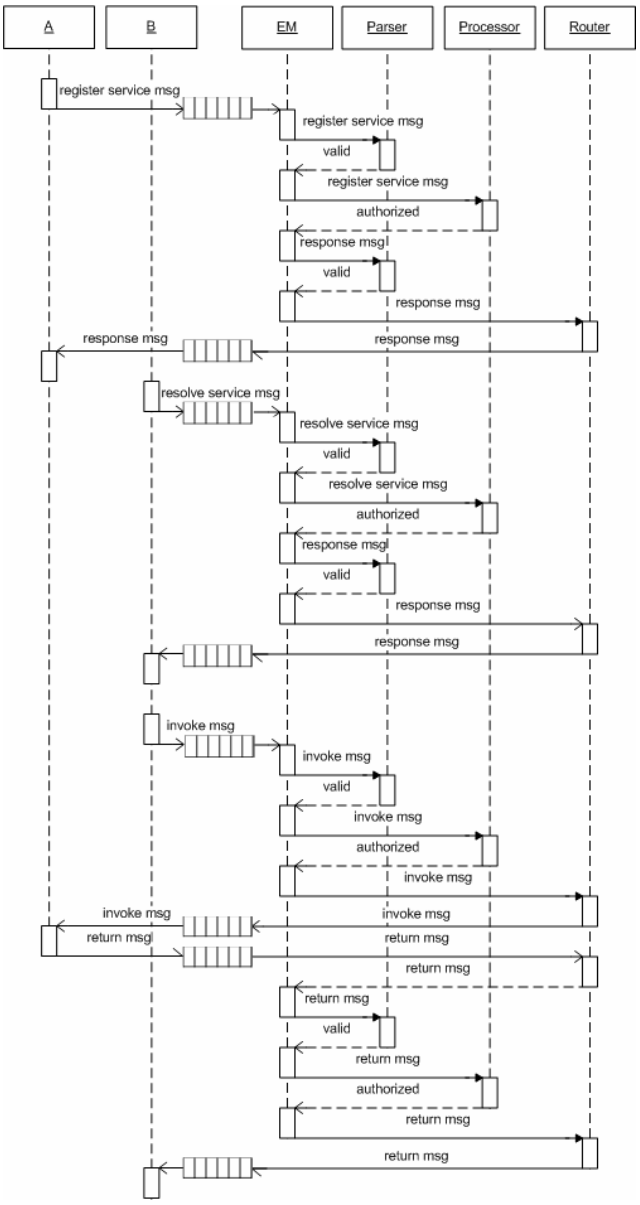

Fig. 3 Service S providing

The first step of the sequence of operations done by A and B is the registration of A as a service S provider in the SSB, by sending an appropriate Service Message to the EM. When EM receives it, EM invokes the Parser to validate the message, and then the Processor to perform access control (this access control aims to verify that A has the permission to offer service S). If the previous steps complete with success, EM stores in LDAP this

registration and assign a specific queue to application A that will use it to listen for requests. EM builds a response message and sends it to A by the Router, after a previous validation.

After this registration step, the application B can invoke the service S, and to do it, it has to send to EM a first Service Message for service S resolving. When EM receives it, EM invokes the Parser to validate the message, and then the Processor to perform access control (to verify that B can invoke service S). When EM sends the reply to B by the Router after a previous validation, the requestor prepares the appropriate message that contains the specific S invocation, and sends it to EM by *DATA_CHANNEL*. After a Parser validation and a Processor access control and informative content translation, EM controls the message and delivers it, by the Router, to the specific queue previously assigned to A.

When A receives a message through its personal queue, analyses it and performs the service, then it builds a response message to be delivered to B, and sends it to EM, that validates and processes it, and then forwards this reply to the invoker B.

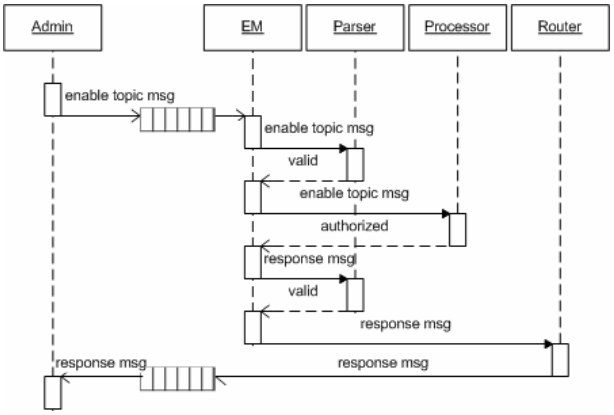

Fig. 4 Admin application enabling topic for event E delivery

4.2 Delivering event E

Let's consider two applications that are plugged into the SSB, and let's call them A and B; a third application, called Admin, is also using SSB functionality. Let's assume that A wants to publish events of type E, and B wants to receive events of type E.

To publish events on a topic, the latter has to be enabled. This first step can be triggered by any application having the permission to do it, for example an administration control service. In this example, the application Admin sends the appropriate Service Message to EM, with the purpose of enabling the topic for event E. When EM receives it, it validates the message by invoking the Parser, and then passes the message to the Processor that performs the access control. Then, EM stores the topic enabling in LDAP, builds a response message, and then validates and sends it to the Admin application.

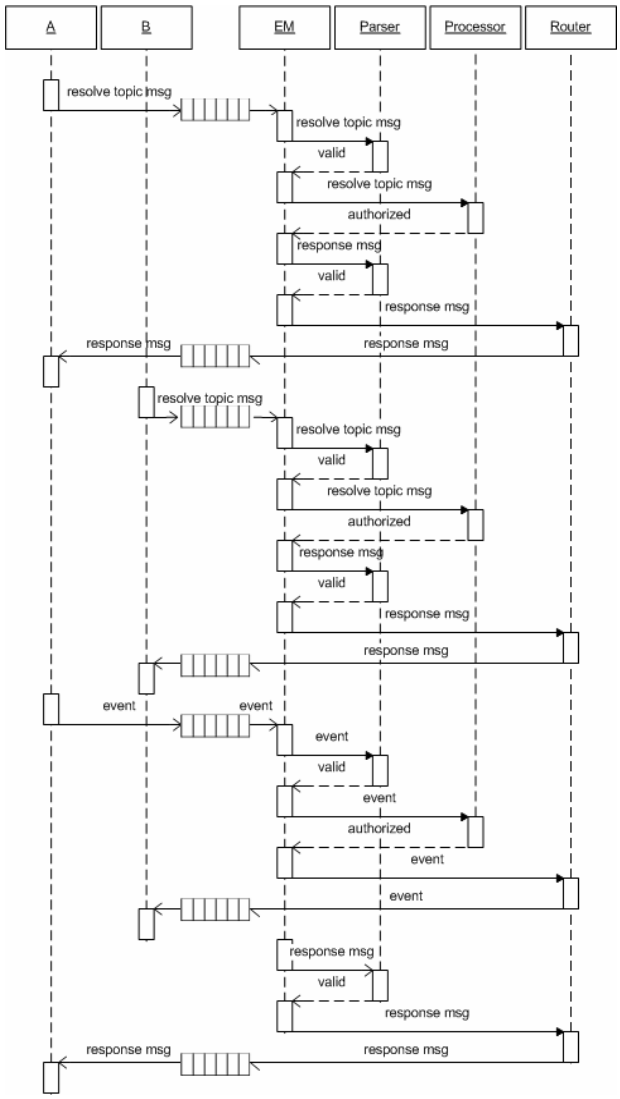

Fig. 5 Event E publish and delivery

Once that the topic has been enabled, A sends a Service Message to EM with the purpose of previously resolving the topic for event E and getting the permission to generate it. When EM receives this message, it validates it by the Parser and invokes the Processor for access control, then EM controls in LDAP that the topic has been enabled previously. EM builds the response, validates it, and then sends it to A.

Also B has to resolve the topic and get the permission to listen for the event E topic. The steps are similar to those described above for application A. When B receives the message from EM enabling it to listen for the event E topic, it starts to listen for this topic for events.

Every time that A wants to send an event, it builds an appropriate Events Message, and sends it to EM by *EVENT_CHANNEL*. When EM receives this message, EM validates it by invoking the Parser and processes it by invoking the Processor for access control, and then forwards the event to the correct topic by the Router. No response is expected from receivers, so EM directly

builds the reply message that notifies to the publisher the result of this operation. This message is validated by the Parser and then sent by the Router to A.

5 Conclusions and future developing

In this paper we have presented our implementation of an ESB, called SSB, which has been utilized to integrate systems of the airport environment. In particular, we have focused our attention on its central layer, the *Enterprise Server Layer*, that provides typical ESB characteristics. We have shown two examples with the purpose of explain how EM works when one or more applications plugged in the bus send a message.

Our future development regards the *service composition*. The main idea is to take advantage of plugged services to provide new services that can be realized by invoking two or more existing services.

These "composed services" have to be based on a context related to the semantic of the entire enterprise system. For this reason, we would define an *ontology*, ("a *systematic, computer-oriented representation of the world*" [15]), a set of relations that is applied to our ontology entities (deontology), and a service specification language, that describes a service from a composition point of view.

References:

[1] Chappel, D. A., *Enterprise Service Bus*, O'Reilly Media, 1st edition, 2004.

[2] Fiorano Enterprise Service Bus, www.fiorano.com/products/fesb/fioranoesb.htm

[3] IBM Websphere Enterprise Service Bus, www306.ibm.com/software/info1/websphere/index.jsp?tab=landings/esb

[4] Sonic Enterprise Service Bus, www.sonicsoftware.com/products/sonic_esb/index.ssp

[5] Artix, IONA Enterprise Service Bus, www.iona.com/products/artix/welcome.htm.

[6] Shaoyun Li, Hongji Yang, Hua Zhou, Building a Dependable Enterprise Service Assembly Line (ESAL) for Legacy Application Integration. In *Proceedings of the 2004 International Conference in Cyberworld (CW'04)*, 2004.

[7] Walsh, A. E., *J2EE*, The McGraw-Hill Companies, Milan, ITALY, 1st edition, 2003.

[8] Terry, S., *Enterprise JMS Programming*, M&T Books. New York NY, 1st edition, 2002.

[9] Thomas, N., Buckley, W., Rise of ESB. In *Business Integration Journal*, Business Integration Journal Press, Dallas, September 2003, pp. 50-52.

[10] Hohpe, G., Woolf, B., *Enterprise Integration Patterns*, Addison Wesley Professional, 1st edition, 2003.

[11] Noffsinger, W.B., Niedbalski, R., Blanks, M., Emmart N., Legacy Object Modeling Speeds Software integration. In *Communication of ACM*, Vol. 41, No 12, December 1998.

[12] XML specs at http://www.w3.org/XML/

[13] Mangano, S., *XSLT Cookbook*, O'Reilly & Associates, Inc., Sebastopoli, CA, 2003.

[14] http://www.openldap.org

[15] Missikoff, M., Navigli, R., Velardi, P., Integrated Approach to Web Ontology Learning and Engineering. In *Computer*, November 2002, pp. 60-63.

[16] Krafzig, D., Blanke, K., Slama, D., *Enterprise SOA : Service-Oriented Architecture Best Practices (The Coad Series)*, Prentice Hall PTR, 2004

[17] Schule, R., Event-driven architecture: The next big thing. Application Integration and Web Services Summit. Gartner, Inc.