ESB - Modern SOA Infrastructure

Tomasz Masternak, Marek Psiuk, Dominik Radziszowski, Tomasz Szydło,
Robert Szymacha, Krzysztof Zieliński, and Daniel Żmuda

Department of Computer Science, DSRG
Al. Mickiewicza 30, 30-059 Kraków
{tomasz.masternak,marek.psiuk,dr,tomasz.szydlo,
robert.szymacha,krzysztof.zielinski,daniel.zmuda}@agh.edu.pl
http://www.soa.edu.pl

Abstract. *The traditional Enterprise Service Bus (ESB) supports Service Oriented Architecture (SOA) providing faster and cheaper integration of existing IT systems. ESB technology has been arisen from such principles like Enterprise Application Integration (EAI), but it is still not sufficient to satisfy SOA requirements with respect to governance. Vendors of ESB did not adopt a single specification and this creates problems with enterprise-wide harmonization of governance process. This is motivation for extended ESB model being Modern SOA Infrastructure. The study introduces extended ESB model, which encompasses among others such aspects like: concept of adaptive ESB, monitoring patterns for distributed ESB environment, adaptive services, and BPEL engine monitoring and governance. This model positions ESB as a intermediary technology for providing core and extended functions, making it easier to use in world-wide IT environments.*

Key words: Enterprise Service Bus, services monitoring, QoS management, services adaptability

1 Introduction

The Enterprise Service Bus (ESB) is a natural evolution and convergence of Enterprise Application Integration (EAI) and Message Oriented Middleware (MOM) implementing the Java Message Service (JMS). It is also inspired by the Web Service technology, leading to the following traditional definition [1]: ESB is an intermediary that makes a set of reusable business services widely available.

The traditional ESB leverages technology advancements in the scope of the presented platforms, in order to establish a distinct form of middleware that can: (i) support Service Oriented Architecture (SOA) integration requirements, (ii) avoid vendor lock-in by utilizing industry standards, and (iii) standardize integration by utilizing broker functions and cross-application communication based on open standards. Some of the early implementations of ESB were constructed around these principles.

As a whole, traditional ESB implementations are divergent from the typical deployment descriptor/packaging standard. This is due to the fact that ESB vendors did not adopt a single specification such as e.g. Java Business Integration (JBI) proposed for Java-based ESB implementation, leading to situations where multiple ESBs are deployed within enterprise boundaries. Some implementations are specific to a business unit or application while others are tasked with external communications. This creates problems with enterprise-wide harmonization when attempting to distribute operations and configuration responsibilities in order to enable collective governance. A good example is the increasing role of OSGi which applies the service orientation principles within the boundary of a single JVM, or the Service Component Architecture (SCA) accepted as a common framework for wiring components together.

In this context the traditional ESB does not satisfy all SOA requirements with respect to governance, regardless of the middleware or technologies used to implement and host services. Expectations in this area are growing as large enterprises have recently begun to exploit the ESB Federation concept. This leads to an extended ESB model, defined as follows [2]: ESB is an intermediary that provides core functions, making a set of reusable services widely available along with extended functions that simplify the use of the ESB in real-world IT environments.

The goal of this chapter is to present the most important aspects of extended ESB as a modern SOA infrastructure and to summarize the work performed in this area by the DCS (Department of Computer Science) team at AGH UST. Extended ESB is investigated in reference to the SOA Solution Stack proposed by IBM. This clarifies the role of integration technologies in SOA-based system construction and deployment. We introduce extended ESB technologies as elements of the ESB compound pattern [3]. The functionality of each element of this pattern is briefly specified, creating a foundation for the more detailed analysis presented in subsequent sections. The concept of adaptive ESB is introduced as a key aspect of the presented study. Detailed analysis of the adaptive ESB requirements leads to specification of its architectural and functional models. The proposed models facilitate definition of mechanisms necessary for implementing adaptive ESB. As these mechanisms cover the most important aspects of ESB governance, they are very much in line with the functionality of extended ESB. Monitoring ESB activity is also described, as it plays a fundamental role in making governance decisions. The proposed monitoring system is very generic and refers not only to JBI-compliant ESB monitoring but also OSGi service tracing. The mechanisms required by adaptability strategies are described in Section 6. A related concept is the construction of adaptive services. Such services, based on the Service Component Architecture, are proposed in Section 7. Suitable extensions of the SCA execution infrastructure are implemented for this purpose. As orchestration processes play a crucial role in the development and deployment of SOA applications, BPEL engine monitoring and governance is also investigated. The proposed solution is presented in Section 8. Finally, since ESB is an inte-

gration technology originating from MOM, the problem of message brokering is analysed.

2 SOA Solution Stack reference model

SOA application development and deployment should be considered in the context of the SOA Solution Stack (S3) proposed by IBM, which provides a detailed architectural definition of SOA split into nine layers. This model is depicted in Fig. 1. Each layer has a logical and physical aspect. The logical aspect includes all the architectural building blocks, design decisions, options, key performance indicators, and so on. The physical aspect covers the applicability of each logical aspect in reference to specific technologies and products and is out of scope of our analysis.

The S3 model is based on two general assumptions:

- The existence of a set of service requirements that are both functional and nonfunctional and collectively establish the SOA objective. Nonfunctional service aspects include security, availability, reliability, manageability, scalability, latency, and the like.
- A single layer or some combination of layers can fulfill specific service requirements and for each layer the service requirements are satisfied by a specific mechanism.

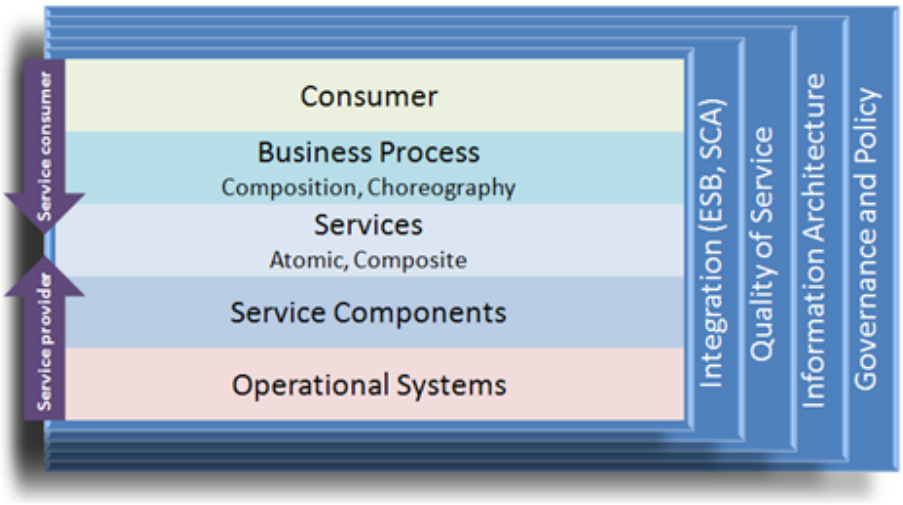

Fig. 1: SOA Solution Stack Model

The nine layers of the S3 stack are as follows: Operating Systems, Service Components, Services, Business Process, Consumer, Integration, QoS, Informa-

tion Architecture, and Governance and Policies. There is no separate layer for business rules. Rather, business rules cut across all layers. The business process and governance layers intersect in defining the rules and policies for a given business process.

A brief characteristic of each layer is presented below as a background for further discussions contained in this chapter.

Operating systems - This layer includes all application and hardware assets running in an IT operating environment that supports business activities (whether custom, semicustom, or off-the-shelf). Because the layer consists of existing application software systems, implementing the SOA solution leverages existing IT assets. Nowadays this layer includes a virtualized IT infrastructure that results in improved resource manageability and utilization.

Service components - This layer contains software components, each of which is the incarnation of a service or operation on a service. Service components reflect both the functionality and QoS for each service they represent. Each service component:

- provides an enforcement point for ensuring QoS and service-level agreements;
- flexibly supports the composition and layering of IT services;
- hides low-level implementation details from consumers.

In effect, the service component layer ensures proper alignment of IT implementations with service descriptions.

Services - The service layer consists of all the services defined within SOA. In the broadest sense, services are what providers offer and what consumers or service requesters use. In S3, however, a service is defined as an abstract specification of one or more business-aligned IT functions. The specification provides consumers with sufficient detail to invoke the business functions exposed by a service provider – ideally in a way that is platform independent. Each service specification must include:

- an interface specification - a collection of operation signatures;
- service endpoint information - the network location to which invocation messages are sent;
- invocation protocol details;
- service semantics, such as measurement units and business context.

It is necessary to point out that services are implemented by assembling components exposed by the Service Component layer. The Service Component Architecture plays an important role within this layer as a promising concept.

Business process - In this layer, the organization assembles the services exposed in the Services layer into composite services that are analogous to key business processes. In the non-SOA world, these business processes are similar to custom applications. On the other hand, SOA supports application construction by introducing a composite service that orchestrates the information flow among a set of services and human actors.

Consumer - The consumer layer handles interaction with the user or with other programs in the SOA ecosystem.

Integration - This layer integrates layers 2 through 4. Its integration capabilities enable mediation, routing and transporting service requests from the service requester to the correct service provider. These capabilities include, but are not limited to, those found in ESB and will be further explained in the following sections.

Quality of service - Certain characteristics inherent in SOA exacerbate well-known IT QoS concerns: increased virtualization, loose coupling, composition of federated services, heterogeneous computing infrastructures, decentralized service-level agreements, the need to aggregate IT QoS metrics to produce business metrics, and so on. As a result, SOA clearly requires suitable QoS governance mechanisms.

Information architecture - This layer encompasses key considerations affecting data and information architectures required to develop business intelligence through data marts and warehouses. The layer includes stored metadata content required for data interpretation.

Governance and policies - The governance and policies layer covers all aspects of managing the business operations' lifecycle. This layer includes all policies, from manual governance to autonomous policy enforcement. It provides guidance and policies for managing service-level agreements, including capacity, performance, security, and monitoring. As such, the governance and policies layer can be superimposed onto all other S3 layers. From a QoS and performance metric perspective, it is tightly connected to the QoS layer. The layer's governance framework includes service-level agreements based on QoS and key process indicators, a set of capacity planning and performance management policies to design and tune SOA solutions as well as specific security-enabling guidelines for composite applications.

3 Extended ESB functionality

The presented S3 model provides a general landscape for the extended ESB functionality specification.

This functionality emerges even more clearly in the context of ESB Federations used by large enterprises. An ESB Federation consists of many ESB instances which are organized into several overarching groups:

- working with service units - focus on loose coupling;
- working with networks - focus on resource virtualization;
- linking services together - focus on agile composition;
- doing 1-3 economically - focus on scalability across several dimensions;
- enabling new offerings based on the new technology - focus on the future.

As enterprises deploy multiple ESB instances, they need to provide an infrastructure to manage, secure, mediate, and govern these instances. The capabilities inherent in an ESB Federation solution are delivered through a standards-based SOA infrastructure. These capabilities include:

- Security - ensuring the privacy, authenticity, authorization, non-repudiation and auditing of all messages moving within and between ESB instances and other service-oriented applications. This also includes decoupling of the security management model from the application programming model;
- Mediation - many applications and ESB instances will use and support different standard protocols and technologies along with different invocation, synchronicity, reliability, and security models. An ESB Federation solution provides policy-based mediation between the various synchronicity, reliability, programming and security models, technologies, messaging styles and standards, ensuring seamless interoperability.
- Management - as ESB instances proliferate, enterprises will require a holistic view of the transactions that traverse their platforms and applications. An ESB Federation solution provides monitoring, management, SLA and alert reporting capabilities for all managed platforms including ESB instances.
- Governance - the ESB Federation solution provides a consistent policy definition, implementation, administration, management and enforcement mechanism for all SOA-enabled platforms and ESB instances within the enterprise. It verifies that all services implement and enforce the same set of policies, and can comply with the policies that will be enforced downstream.

The ESB Federation solution is an infrastructure solution. It is not an ESB for multiple ESB instances - rather, it simply provides core policy-based infrastructure services for heterogeneous ESB instances. A successful deployment of the ESB Federation requires the presence of suitable capabilities in each ESB instance.

The list of core functions which provide the basic operational capabilities of the ESB is rather long:

1. Support for multiple protocols;
2. Protocol conversion;
3. Data transformation and data-based routing;
4. Support for multiple connectivity options;
5. Support for composite services through lightweight orchestration;
6. Support for multiple standard business file formats;
7. Integrated security features;
8. Comprehensive error handling mechanisms;
9. Support for both synchronous and asynchronous operations;
10. Highly available and scalable infrastructure;
11. Support for many options in each of the above categories;
12. Extensibility.

Extended functions are those ESB capabilities that lie beyond the operational machinery listed above. They make services widely available to consumers and offer broad support for SOA application development, provisioning and running. Development of these extended functions is currently the focus of attention by ESB vendors. Fig. 2 summarizes the most important extended functions.

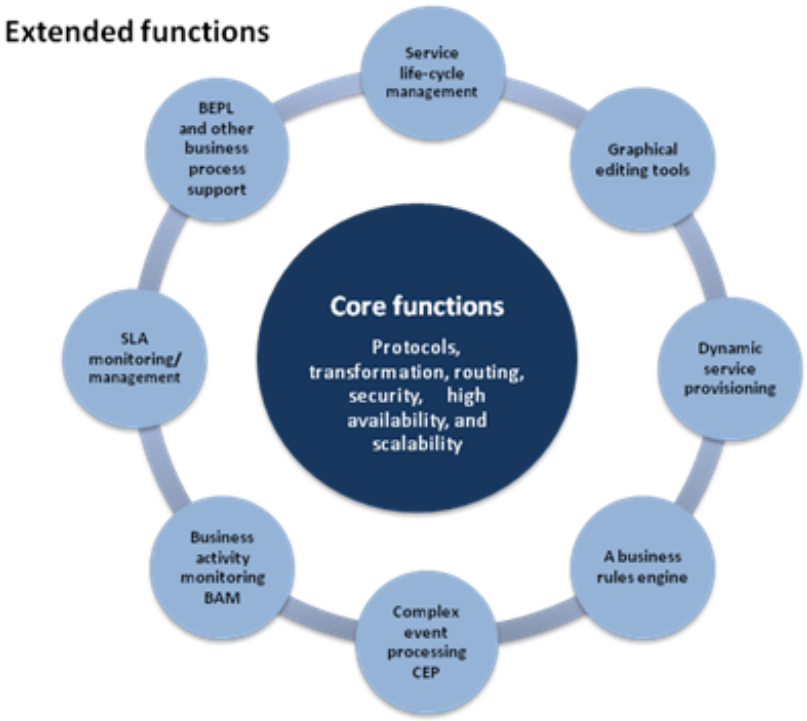

Fig. 2: Core and extended ESB functions

A more detailed specification of these functions is presented in Table 1. It is necessary to point out that many related requirements are still not fully satisfied and require further research. The following sections elaborate on this issue in more detail.

From the software engineering point of view ESB can be treated as a Compound Design Pattern which in turn represents collection-related patterns. The relationship between these patterns is immaterial as compound patterns focus on results of their combined applications. A compound pattern represents a set of patterns that are applied together to a particular application in order to establish a set of design characteristics.

Fig. 3 shows the ESB compound design pattern. It is fairly complex as it comprises other patterns (such as the Service Broker) which are also compound patterns. Moreover, the ESB pattern can be considered an element of the Canonical Schema Bus. Patterns connected by solid lines represent the core functionality of ESB, while remaining patterns address extended ESB capabilities. The functions of extended ESB, as shown in Table 1, cover the orchestration process. This leads to the conclusion that the extended ESB compound pattern should be

Function name	**Requirements**
Graphical editing tools	Graphical editors for ESB flows such as itineraries or lightweight orchestration.
SLA monitoring and management	Reporting and registration of QoS and QoE. Load balancing to meet SLA. Endpoint security and management.
BPEL and other business process support	Design, simulation, and execution of business processes using BPEL.
BPEL engine activity monitoring	Business activity monitoring (BAM) Definition of business-centric metrics (KPI) and their presentation in near-real time. Generating alerts when KPIs cross specified thresholds.
Service lifecycle management	Service definition, implementation and installation.
Dynamic service provisioning	Dynamical provisioning of new ESB operations. Modification of flows without restarting ESB components. Dynamic control over the number of running service instances in accordance with SLA targets.
Complex event processing (CEP)	Ability to define and recognize complex events during business process execution.
Business rule engine (BRE)	Ability to specify and impose policies related to business or system activity. Easy integration with CEP and EDA (Event-Driven Architecture).

Table 1: Extended ESB functions

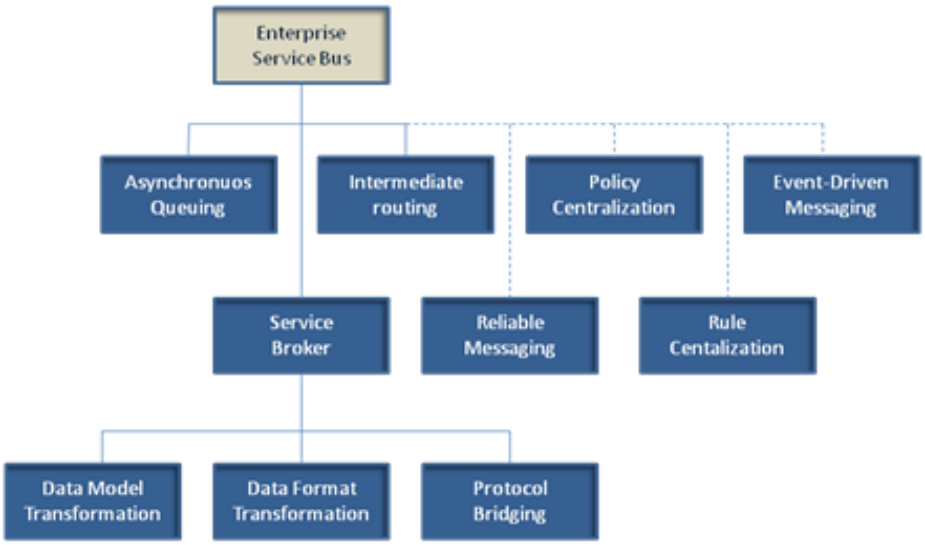

Fig. 3: ESB compound design pattern

further enhanced with Orchestration compound patterns, which play a crucial role in SOA.

4 Model of adaptive ESB

The detailed analysis of SOA infrastructures presented in the previous sections shows that its elements should offer some level of adaptability. In fact, adaptability looms as a dominant aspect, cutting across all extended ESB functions. An *S3 Layer* is defined as a set of components, such as architectural building blocks, architectural decisions and interactions among components and layers. This definition emphasizes the existence of many options that can be subjected to architectural decisions taken during the SOA application design phase or postponed until runtime. A generic structure of an adaptive ESB functional element is shown in Fig. 4. The policy engine perceives the system as an abstraction exposed by the exposition layer and is able to perform a set of actions on this abstraction. The abstraction can be defined as a complex service execution model, or a simple set of services involved in execution. In both cases an adaptation policy has to be defined. This policy is used to represent a set of considerations, guiding decisions. Each adaptation policy may express different goals of system adaptation, such as minimizing system maintenance costs or ensuring particular QoS parameters.

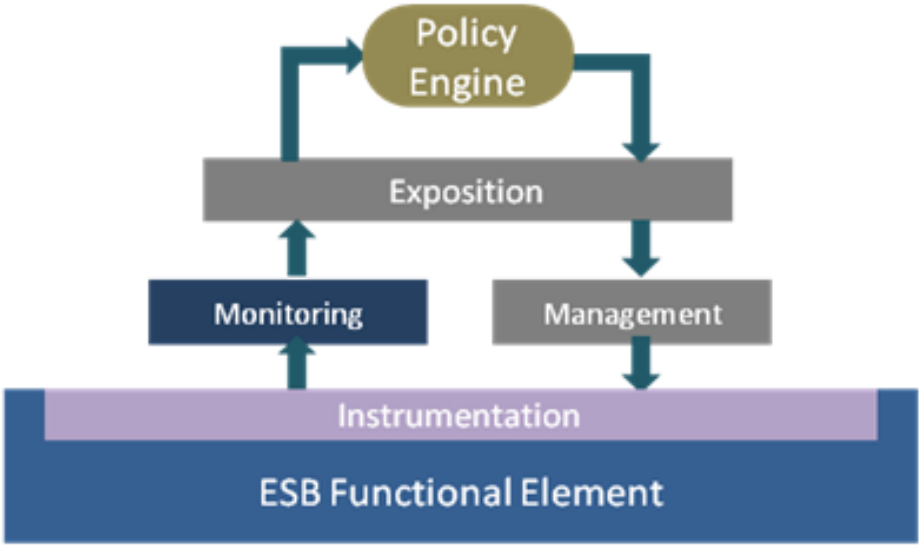

Fig. 4: Adaptive ESB functional elements

Design-time decisions cannot, however, take into account every context of service execution and lower-layer architectural configuration. Hence, runtime

architectural decision processing is particularly important for the S3 stack. Each layer must therefore be equipped with suitable adaptability mechanisms enabling the enforcement of these decisions.

System Elements

The Adaptive ESB functional element is constructed around the well-known closed loop control schema. A fundamental part of this element is the adaptability loop which consists of the following blocks:

- **Instrumentation** – the Enterprise Service Bus is enriched with additional elements that gather monitoring data from the ESB and perform adaptability operations;
- **Monitoring** – determines system state and traces monitoring service invocations. The stream of monitoring data is processed by the Complex Event Processor and listeners are notified of complex monitoring events. The monitoring layer exposes sensors for the *Exposition layer*;
- **Management** – components used to manage the execution of services deployed in the ESB and for lifecycle management of interceptors. The management layer exposes effectors for the *Exposition layer*.
- **Exposition** – components that expose the state and events occurring in the ESB as facts for the *Policy Engine Layer*. This layer also contains components used to model system behavior in the model domain;
- **Policy Engine** – rule engines responsible for service adaptation in order to meet high-level goals defined as policy rules.

A number of distributed adaptive elements can be managed by a global Policy Engine in accordance with a high-level strategy. The instrumentation layer enriches the ESB with additional elements. These elements provide adaptability transformations necessary to achieve adaptive ESB and are described in the following sections. The monitoring layer is responsible for supplying notifications of events occurring in the execution environment. As the volume of monitoring information gathered from the ESB could overwhelm the *Exposition layer*, events are correlated with one another and notifications are sent only about complex events. Complex Event Processing can be compared to an inverted database containing stored statements: as data arrives in real time, these statements are executed and notifications are sent to registered listeners. The policy engine layer analyses facts and infers decisions which are then implemented in the execution environment. Facts representing the state of the system or events occurring in the system are supplied by the exposition layer.

The approach presented in this section is a model-driven adaptation policy for the SOA. The system analyzes composite services deployed in the execution environment and adapts to QoS and QoE changes. The user composes the application in a chosen technology and provides an adaptation policy along with a service execution model. The architecture-specific service composition layer continuously modifies the deployed service in order to enforce the adaptation policy.

The abstract plan, providing input for architecture-specific service composition, can be hidden and used only by IT specialists during application development. System behavior is represented by the service execution model.

The composite service execution model is an abstraction of the execution environment. It covers low-level events and relations between services and exposes them as facts in a model domain for the *Policy Engine*. Decisions taken in the model domain are translated to the execution environment domain and then executed. The *Model Analyzer System* gathers monitoring data from the execution environment via monitoring and management agents.

This process can be described as architecture-specific service composition adaptation that is performed in order to achieve the required value of QoS or QoE guided by the adaptation policy. The adaptation loop addresses service selection, binding protocol and interaction policy choices. Thus, the adaptability process mainly concerns integration mechanisms.

5 ESB monitoring

The service orientation paradigm enables IT architects to identify independent parts of business logic and expose them as services. Additional service design methodologies [4] classify services into types and hierarchies, stressing the most important service attribute: composability. The composability of services significantly increases the potential size to which a fully operational service-based enterprise system can be extended. In order to provide sufficient adaptability to constant changes in business requirements occurring in such large and complex infrastructures, proper monitoring mechanisms have to be introduced. The Enterprise Service Bus is the most frequently used design pattern for enterprise service backbones. Therefore, ESB monitoring is a vital element of service-oriented IT systems and is essential for the Adaptive ESB Model (cf. Figure 4). This section defines the goal of ESB monitoring and presents the problems which have to be solved in its context.

ESB monitoring background

Our study focuses on the SOA paradigm realized using Java-centric technologies; therefore we will focus on OSGi [5] and Java Business Integration (JBI) [6]. JBI is a Java-centric standardization of the ESB pattern while OSGi introduces a dynamic service-oriented modularization model in the context of a single JVM process. OSGi adds significant flexibility by transforming the flat dependency space into a well-structured hierarchical one and by introducing dynamic single-process service repositories. OSGi is often referred to as single-process SOA [7]. The JBI specification has not been widely adopted [8] and currently provides a point of reference rather than an actual ESB implementation standard. Modern ESB infrastructures tend to take OSGi as the basis for single-process SOA and extend it, by means of federations [9], to a distributed service backbone. Therefore our study begins with a definition of OSGi monitoring and assumes JBI

as a point of reference in the construction of monitoring mechanisms for ESB Federations. Figure 5 depicts the structure of monitoring in a service-oriented environment. Monitoring is divided into domains and perspectives. The infrastructure domain refers to monitoring of the system environment with focus on attributes of the operating system and any virtualization layer between the OS and actual hardware. The platform domain relates to the platform on which the container of a higher domain is executed. JVM monitoring is the subject of the JVM platform domain on which this study focuses. The container domain is the actual element where the system part of OSGi and ESB monitoring is located. The monitoring of application-related service attributes takes place in the application domain and further evolves into monitoring of important Key Performance Indicators [10] in the business domain.

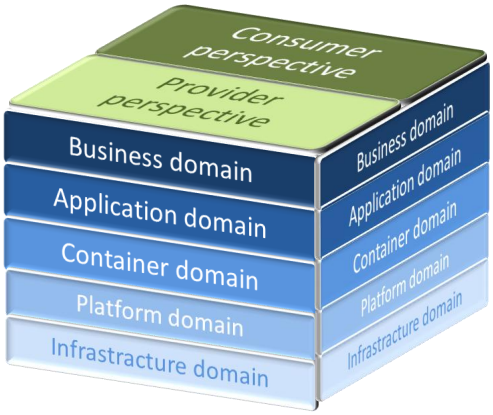

Fig. 5: Monitoring layers in a service-oriented environment

Thus ESB-based SOA monitoring spans the following domains: container (system monitoring), application (application monitoring) and business (business activity monitoring (BAM) [11]). In all domains monitoring can be performed either from the consumer's or provider's perspective. For ESB monitoring, both perspectives are important. The presented monitoring structure can be mapped onto layers of the S3 model (cf. Figure 1). The container domain corresponds to S3 Service Components and Integration, while application and business domains are counterparts of S3 Services and Business Processes.

The goal of the presented ESB monitoring is to provide a complete solution for monitoring of both perspectives in the three ESB domains. The functional requirements of ESB monitoring can be summarized as follows:

– Gathering complete information on the OSGi and ESB level: monitoring service deployment – current service topology as well as service invocation parameters.

- Applying a design which enables transformation of system-level monitoring information with no semantics into enriched information providing business value.
- Ensuring flexible, declarative monitoring target specification, which can be changed at any time.
- Monitoring should adapt to changes in the distributed ESB environment in order to ensure fulfilment of the defined goals.

In addition to functional requirements there is also a set of nonfunctional design assumptions which influence the architecture to a significant degree. These assumptions are tightly related to the *monitoring scenario* concept, which specifies the desired monitoring outcome from the user's point of view (the following section provides more details about *monitoring scenarios*). The nonfunctional assumptions are as follows:

- Monitoring has to be selective: only information required to implement *monitoring scenarios* at a given should be gathered.
- Monitoring mechanisms should not influence the performance of elements which are not currently monitored and any performance deterioration of monitored elements should be minimal.
- Monitoring data should be propagated only to locations which are needed to realize a given *monitoring scenario*.
- System reconfiguration in response to changes in *monitoring scenario* definitions should not take longer than several seconds.

These requirements outline a large problem space which has to be explored. The most challenging aspects are selectivity and business activity monitoring. In order to satisfy selectivity requirements, the described approach introduces a methodology for mapping the *monitoring scenario* to a proper configuration of *interceptors* (cf. next section) which collect monitoring data. The first attempt at implementing BAM focuses on enriching the *monitoring scenario* with a business context and introduces *interceptors* capable of providing information with business value.

Monitoring model

The general outline of ESB and OSGi monitoring, described in the previous section, introduces the need for *monitoring scenarios*. To describe such scenarios in compliance with all stated requirements the following aspects of ESB and OSGi monitoring have to be considered:

- topology monitoring - refers to monitoring service topologies, gathering information about topology changes in federated containers;
- parameter monitoring - gathering the values of predefined metrics such as performance, reliability, composability etc. Each metric installed in any of the containers introduces a set of attributes which have to be defined in order to gather monitoring data;

– content monitoring - refers to monitoring particular service invocation parameters or business application errors.

In order to specify a *monitoring scenario* two steps have to be performed. First, the topology of the monitored container must be acquired. It is impossible to specify a *monitoring scenario* without proper information about its current topology state. In accordance with these assumptions, a scenario definition encompasses topology elements for which the scenario will be applied. Such a specification may be represented by a set of patterns, matching particular topology elements to their names. The second step of scenario specification is to define a metric which will be applied to the predetermined topology elements. Each metric has an individual set of parameters. Information about the values of selected parameters has to be included in the *monitoring scenario* definition. One monitoring scenario can contain many metrics and topology elements – each pair is called a *monitoring subscription*.

Having defined a *monitoring scenario*, we need to specify a mechanism capable of flexibly plugging into the monitoring logic of OSGi and ESB. The proposed model introduces a non-invasive mechanism called the *interceptor socket*. In both cases, such a component must provide functionality for plugging interceptors and providing them with monitoring data. The interceptor socket is an element which enables the monitoring system to be dynamically connected to the monitored environment. It seems reasonable to state that in the ESB environment interceptor socket mechanisms have to be capable of processing messages passed to services installed in a container. In the latter case we need to ensure that service invocations are intercepted along with their context (i.e. invocation parameters). Such a unified approach for both OSGi and ESB containers enables dynamic management and installation of interceptors. As can be seen in fig. 6, interceptors installed in sockets feed monitoring data to higher-level monitoring logic through the monitoring backbone. This backbone is one of the core components in every distributed monitoring system and can be the subject of extensive analysis.

In order to describe the possible methods of creating interceptor chains it is first necessary to describe what an interceptor is. The presented study introduces the concept of Interceptor as a Service (IaaS). The OSGi-based IaaS realization facilitates reconfiguration and management of interceptors while fulfiling one of the most important nonfunctional assumptions, i.e. selectivity of the monitsoring model. Service-oriented interceptors can be divided into two groups: agnostic and business interceptors. The former category defines interceptors which do not need to analyze invocation parameters (content), but instead focus on information gathered from the invocation context (e.g. the processing time of a particular service). The latter category defines business interceptors which analyze content (service parameters in OSGi or message content in ESB) and process such data in accordance with the implemented business logic. These two categories are organized into separate chains. Each chain is an ordered list of interceptors with a defined sorting strategy. It determines the order in which interceptors process data and pass monitoring results to the backbone.

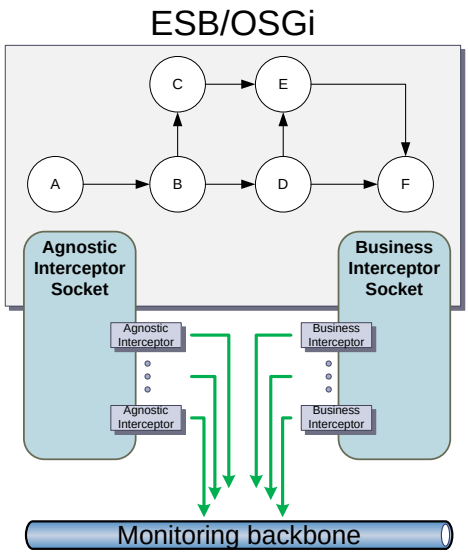

Fig. 6: Connection between the monitored environment and the monitoring backbone

The presented interceptors need to be created and configured in accordance with *monitoring scenario* details. *Monitoring subscriptions* of the current scenario are installed in the monitoring environment and serve as a basis for the creation and configuration of interceptor chains. Special interceptor configurations can cover services which do not exist at subscription time. If a new service is activated and matches a given topological pattern, there is no need for reconfiguration. Such a service can be automatically monitored in accordance with specified metrics. *Monitoring scenarios* allow flexible transformation of business domain monitoring goals into low-level system monitoring configurations. The business analyst has to specify a set of *monitoring scenarios* which have to be implemented in order to extract business domain data from system events. Those scenarios, along with sets of monitoring subscriptions, are propagated throughout the distributed ESB.

The most important (and most problematic) aspect is the distribution of specific *monitoring scenarios* in the distributed ESB environment. They not only have to be periodically propagated (in order to publish changes), but also synchronized to maintain the consistency of *monitoring scenario* definitions. On the other hand, as described in the previous section, the solution must ensure *selectiveness* of monitoring capabilities. Fulfiling those goals is not an easy task and must be approached with care. Those aspects, along with the monitoring backbone design architecture (described earlier), will be the main direction of future work originating from the presented study and aiming at implementation of BAM in a Federated ESB environment.

6 ESB adaptation mechanisms

Monitoring ESB allows management actions to be performed. The presented concepts cover selected issues related to improving communication over ESB by separating traffic into several flows, which can then be independently handled. This leads to increased scalability and is required by many aspects of infrastructural functionality, including monitoring, data collection, management information distribution and security considerations.

The proposed mechanisms are compliant with existing integration technologies. Interceptor mechanisms enable dynamic control of service or component invocation, along with sensors and effectors necessary to close the adaptation loop.

As mentioned earlier, the Enterprise Service Bus is an integration technology that allows architects to compose applications with services using various communication protocols. ESB provides mechanisms for message normalization and routing between selected components. The proposed adaptation mechanism provides elements necessary to close the control loop for ESB. It is used for compositional adaptation in systems which modify service composition while retaining their overall functionality. ESB is suitable for implementation of such adaptation, since one can modify message flows between components in accordance with high-level goals.

Sensors

An adaptive system should adapt itself in accordance to various goals, requiring several types of information from ESB. In most cases, this information will be disjunctive, so one would expect to deploy specialized types of sensors rather than generic ones. Interceptor design patterns fulfil these requirements, allowing interceptors to be deployed or undeployed at runtime.

A message is created in a service, passes through the specialized Service Engine and is sent to the Normalized Message Router (NMR), which reroutes it to a particular destination through the same components. Common usage of this concept includes:

- **QoS Measuring** – the functionality of monitoring interceptors is not limited to creating copies of messages sent through them, but may include more complex tasks, providing quantitative information related to service invocation. In the QoS interceptor, a message sent to a service is stored in the internal memory of the interceptor. When a response is generated, the previous message is correlated and response time is evaluated;
- **Tagging/Filtering** – interceptors are commonly used for message tagging and filtering;
- **VESB** – it is possible to implement the Virtual ESB (VESB) pattern which can be compared to solutions currently used in computer network traffic management, such as VLAN[12].

Effectors

Adaptive software has to adapt itself to changes in the execution environment. This adaptation requires performing actions (via effectors) that modify the execution characteristics of a system. For any sort of system, a set of operations has to be defined, along with places where these modifications should be introduced. It has been found that modifying message routes can affect the adaptation of complex services deployed in the ESB.

Rerouting messages to other instances is justified only when these instances share the same interface and provide identical functionality. Current implementations of ESB that are compliant with the JBI specification share some common attributes used in the course of message processing. Each invocation of a complex service is described by its CID (Correlation ID) and is constant for one invocation, even when passed among services. A particular message sent between services is described by an EID (Exchange ID). Generally speaking, the invocation of a complex service is described by a CID and consists of several message exchanges described by an EID. Extending the Normalized Message Router with a routing algorithm able to modify message routing would yield an adaptive ESB.

Routing Algorithm Once the message reaches the NMR, the routing algorithm relies on matching the message to routing rules in the routing table. If the message matches a routing rule, that rule is fired and the Service Name from the routing rule substitutes the intended destination Service Name. Routing rules are split into groups with different priorities, analyzed in a particular order. In every message, parameters such as VESB tag, Correlation ID, intended Service Name and Exchange ID are matched to routing rules. If the message matches several rules, one of them is selected on a round-robin basis to provide load balancing.

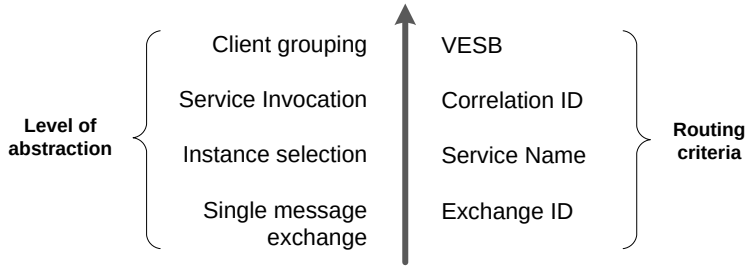

Fig. 7: Abstraction level of message flow

Rule priorities are assigned in accordance with the abstraction levels depicted in Fig.7. Rules are analysed in the order of increasing priority values.

Priority	VESB	CID	Intentional Service Name	EID	Decision
1	n/a	n/a	n/a	+	Service Name
2	n/a	+	+	n/a	Service Name
3	+	n/a	+	n/a	Service Name
4	n/a	n/a	+	n/a	Service Name
5	n/a	+	n/a	n/a	Service Name
6	+	n/a	n/a	n/a	Service Name
7	n/a	n/a	n/a	n/a	Service Name

Fig. 8: NMR routing table

Depending on the priority value, different parameters are matched. The presented matching criteria are summarized in Fig.8. The lower the priority value the more important the routing rule. Some attributes are omitted when processing rules, although each routing rule returns a Service Name as a result. The decision element which closes the adaptation loop presented in chapter 2 gathers information about VESB from sensors and uses effectors to dynamically modify the routing table at runtime.

7 Adaptive Components

Adaptive Components are used in the extended ESB model to provide adaptability for the Service Component layer, as described in the SOA Solution Stack model. The main goal of Adaptive Components is to provide a background for creating atomic services used by upper layers.

In SOA systems, atomic services are mainly comprised of components. The need to ensure implementation and communication protocol independence forces the selection of an independent integration environment. The Service Component Architecture (SCA) specification [13] provides such a solution. It supports several component implementations (such as Java, C++, BPEL and Spring) as well as various communication protocols (Web Services, RMI and JMS). Moreover, it is highly extensible, as evidenced by different SCA implementations [14] which introduce additional technologies not covered by the original blueprint.

It should be noted, however, that the SCA specification lacks adaptability mechanisms which play a crucial role in ensuring conformance between the provided services and changing user requirements (i.e. Quality of Service). This section describes adaptability mechanisms introduced into the SCA specification.

SCA Adaptability Requirements

The SCA specification involves the concept of a composite which consists of a set of components connected by wires. A wire consists of a service representing the functionality exposed by one component, a reference representing the functionality required by another component (delegate object) and a binding protocol

which represents the communication protocol between the reference and the service. Several components in an SCA composite may expose services for external communication. Such services are further exposed e.g. via ESB.

Services created using SCA composites should be able to adapt to changing customer requirements and service provider capabilities (such as different CPU speeds resulting from infrastructure changes etc.) Non-functional customer requirements are recognized as QoS metrics and may be provided by the QoS layer of the S3 model. On the other hand, the measured service parameters are called Quality of Experience (QoE), and represent actual capabilities of a service. In an ideal case, QoE metrics should be as close to QoS requirements as possible.

The presented discussion leads to augmenting component descriptions with additional information about the provided QoS and monitoring of actual QoE values during runtime.

Service Model

An SCA composite may be perceived as a directed graph with nodes represented by components and edges represented by wires (directed from references to services), further called a *Composition Instance* (CI). A CI is connected to a specific QoE description derived from a composite. A set of *Composition Instances*, which meet the same functional requirements and differ only with respect to nonfunctional ones (QoS) is called a *Composition*. *Compositions* may also be represented as directed graphs, created by joining all CIs which provide a given element of functionality. The rules for joining CIs are as follows: if CI1 and CI2 use the same component as a specific node, they can be joined; otherwise both need to be added to the graph. Such a solution reduces the amount of resources required by all CIs (by exploiting shared nodes) and enables more efficient CI processing. A sample *Composition* with a selected *Composition Instance* is depicted in fig. 9.

Adaptation Mechanisms for SCA

Adaptation of services consisting of components may be perceived as selection of a proper Composition Instance from those available, in accordance with the stated QoS requirements and observed QoE values. This leads to a classic adaptation loop which includes monitoring, planning and selecting proper Composition Instances at runtime.

The SCA specification and its existing implementations do not provide adaptability mechanisms. Likewise, dynamic wire changes are not allowed. To provide support for changing wires at runtime an interceptor pattern may be used. Once intercepted, invocations between components may be sent via wires, as determined by given adaptability policies. Such interceptors may also be used for monitoring purposes, exposing data flows to other layers in the S3 model. The SCA monitoring and management functionality is exposed via a specialized Monitoring and Management Interface, further used by the inferring components, as

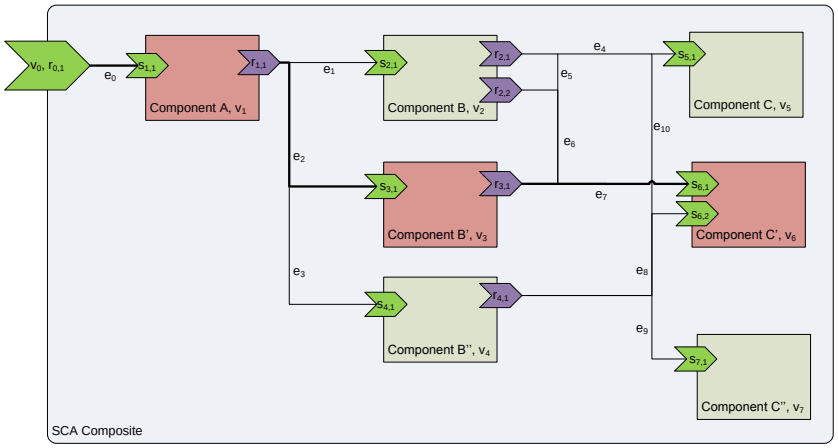

Fig. 9: Sample Composition with a selected Composition Instance

described below. A sample Adaptive Component built using enhanced SCA is depicted in fig. 10

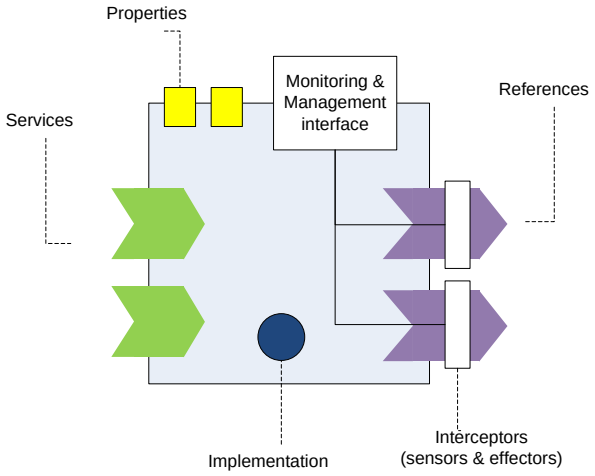

Fig. 10: Adaptive SCA Component

Selecting a *Composition Instance* which best fits the stated nonfunctional requirements may be a very complex task, due to the large amount of existing components (the number of possible *Composition Instances* rises exponentially along with the number of existing components in a *Composition*) and the large set of rules. Thus, advanced rule engines (RE), such as Drools [15] or JESS [16],

can be used. Applying REs to selecting CIs provides a policy enforcement point and simplifies the S3 Policies layer.

Further Work

Adaptive Components are a basis for further work with the aim of enhancing existing component descriptions to automatically generate *Compositions* based on semantic descriptions. Such descriptions should contain information about functional component capabilities, similar to the OWL-S [17] standard used in the Web Services environment. Starting with a custom-defined goal, a service should be built from components and further adapted using the mechanisms described above.

8 BPEL monitoring

The IBM S3 model [18] places business processes near the top of the stack, in direct support of customer needs. Such processes hide the complexity of lower-level services and components. According to the Workflow Management Coalition [19] a business process is a set of one or more procedures realizing common business aims or policies, usually in the context of an organizational role and dependency structure. BPM (Business Process Management) [20] systems usually exploit the declarative system functionality construction model. They determine aims but do not state how to achieve them. Some notations, such as BPMN v1.0 (Business Process Management Notation), are used to describe the entire process in abstract terms, while others, like BPEL (Business Process Execution Language), BPELj (Business Process Execution Language for Java), XPDL (XML Process Definition Language) or jPDL (Java Process Definition Language), can be automatically run in execution environments called business process engines [19][20][21]. Currently, the most widely supported business process definition language seems to be BPEL, with more than ten commercial and open-source process engine implementation. As SOA orchestration processes are gaining importance, the existing monitoring and governance tools for vendor-specific BPEL engines lag behind. Therefore, an experimental prototype of a common multi-vendor distributed monitoring tool has been designed, implemented and validated.

Existing BPEL design, monitoring and management tools

Existing BPEL execution environments are augmented with sets of tools simplifying the creation, deployment and management of BPEL processes. Most such environments are based on Eclipse or NetBeans IDEs, with some commercial products claiming to enable process definition over web interfaces. (Results generally correspond to product prices.) Even though each process definition tool is dedicated to a particular process execution engine, they generate BPEL-conformant process definitions which may be run on almost any engine. In the

area of BPEL process monitoring and management the situation is somewhat more complex. Almost each engine provides basic business process monitoring and managements capabilities, sometimes exposed via standardized application GUIs (e.g. BPELMaestro by Parasoft, WebSphere by IBM) and sometimes exploiting vendor-specific APIs (Oracle, ApacheODE, Glassfish). The provided monitoring consoles are not only vendor-specific but also very inconvenient and unwieldy. They only expose basic information about the running process instances and steps (activities) which the processes actually perform. They rarely support filtering, data aggregation and bottleneck analysis, even on a basic level. The following implementations have been investigated in the context of process monitoring and management: BPEL Process Manager by Oracle, WebSphere by IBM, ActiveVOS by Active Endpoints, LiquidBPM by Cardiff, BPELMaestro by Parasoft, bizZyme by Creative Science Systems, Biztalk by Microsoft, ApacheODE by Apache nad jBPM by jBoss. The usability of custom monitoring GUIs is, at best, disputable. All are limited to parsing data provided by their own engines; moreover, they do not enable correlations of subprocesses with base processes and expose only simple query mechanisms based on pull interfaces (if an API is available at all). Often there is no direct support for selection of monitoring data (only an on/off monitoring switch is exposed, with all the efficiency drawbacks resulting from collection and storage of unneeded monitoring data).

Motivation

Current BPEL monitoring tools do not fully meet the stated expectations: the information they provide means nothing for business users and is insufficient for IT specialists. They have to be rethought and redesigned to address the needs of particular user groups. A BPEL process definition is more than a simple blueprint for orchestrating process components - it contains process variables and structure activities that have business names and meanings. Such metainformation, combined with process monitoring data, can be presented to non-technical business users able to validate process correctness from the business/customer perspective. This goal cannot be achieved with ESB and OSGi monitoring, operating at a lower level and presenting complex data in a manner understandable for IT specialists. In order to efficiently monitor a BPEL process, specific monitoring data has to be collected during the entire (usually very long) process execution cycle. This data should include information about individual actions (steps) executed by the process (e.g. execution of a Web Service method, sequence of operation invocations, conditional instructions, exception handling, etc.) together with execution time, additional parameters (e.g. Web Service address), status (e.g. correctly completed, faulty or having generated an uncaught exception) or even the values of BPEL process variables in the context of a particular activity. All this information needs to be collected, filtered, stored, aggregated and presented in a clear and user-friendly way, preferably not specific to any BPEL engine. As none of the existing tools support such functionality, a new, extended, multi-vendor, distributed BPEL monitoring tool is needed.

System Architecture

The architecture of proposed BPEL monitoring system is presented in Fig. 11. Data acquisition is realized by two cooperating modules instantiated for each monitored engine: Monitoring Event Manager and Monitoring Event Emitter. Together, they are responsible for subscription management and data propagation. Monitoring data is obtained from BPEL engines by the instrumentation module or via the engine API, depending on its availability and functionality [22]. The BPEL Management Agent is the main system component, exposing an interface for management of multiple engines, enabling monitoring data filtering, exposing process metadata and ensuring historical data access and registration for monitoring event notifications generated by the BPEL Monitoring Agent. The responsibility of the BPEL Monitoring Persistence Agent is to store and query monitoring data and metadata along with the current tool configuration.

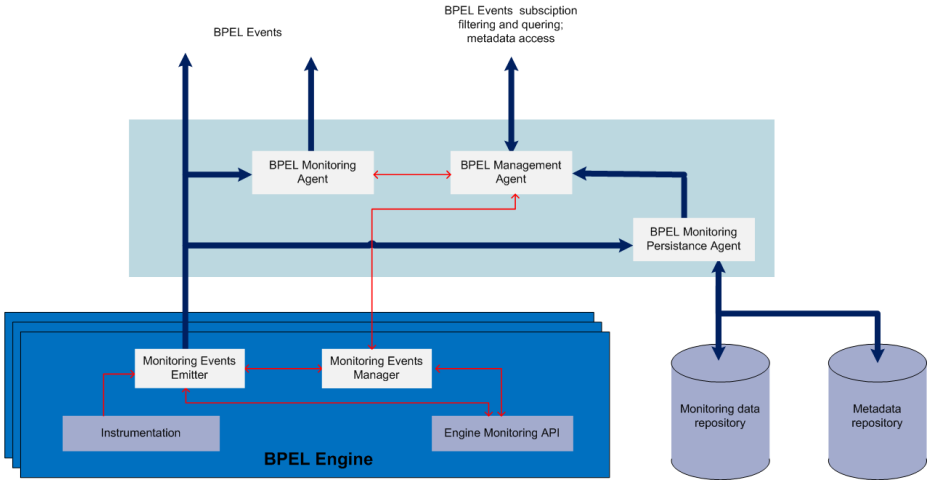

Fig. 11: AS3 - BPEL Monitoring architecture

The proposed BPEL monitoring concept fills, in a natural way, the gap between current BPEL monitoring tools and existing, extremely expensive commercial BAM (Business Activity Monitoring) solutions. BPEL process monitoring must not ignore business users as it does today. It has to shift from measuring low-level, process-specific metrics of BPEL engines (execution time, number of executed process or faults) to more business-oriented goals (number of processes with specified business constraints such as the number of orders with a total amount greater then 100$). The monitoring tools need to enable business users to trace BPEL process execution, basing not only on the status of a process but also on its variables and parameters. This is the challenge we intend to overcome.

9 Messaging

The Enterprise Service Bus, currently recognized as the foremost SOA infrastructure implementation, builds upon the concept of message-oriented communication, i.e. JMS. Asynchronous communication inherent in MOM solutions is a key enabler for loose coupling and autonomous operation - basic tenets of service orientation. In the early implementations of ESB only basic mechanisms provided by the messaging infrastructure were used. Newer approaches extend the set of basic capabilities, taking advantage of QoS assurances provided by messaging solutions.

Communication models and routing

The most frequently used communication model assumes a single recipient. In such a case, asynchronous message queuing is used, where each participating side reads incoming messages from a local queue and processes the received data. Message Exchange Patterns extend this basic approach by defining interaction schemes in which correlations between groups of messages are introduced. As a result, even though there is no direct support for synchronous communication, it can be applied when necessary. Single-receiver delivery is one of two commonly available possibilities, the other one being the publish-subscribe model. In this latter case, each message sent by a single publisher is received by a group of clients which share common interest in the data being produced. The notion of a topic and optional peristent data delivery discussed in the following paragraphs provides a high degree of decoupling both in space and time domains.

In addition to basic addressing schemes, more complex mechanisms are available. JMS extends basic message routing with the notion of message selectors. In such a case messages are routed on the basis of message header properties. Furthermore, some currently available platforms provide mechanisms for content-based routing, heavily used in EIA. All of the above mentioned mechanisms can be used in conjunction with one another. Multipoint data delivery is very crucial from the EDA perspective, where events produced at a source have to be delivered to all interested parties. Different types of information used for routing purposes are presented in Fig. 12.

Apart from the basic functionality, message-oriented middleware provides more advanced capabilities in the form of QoS assurances, summarized in Table 2.

Message Oriented Middleware

Message oriented middleware, as a core part of the SOA infrastructure, plays a key role in the scope of nonfunctional system properties. In order to enable advanced functionality, i.e. 5, the communication infrastructure should be appropriately constructed. From the system's point of view, MOM can be seen as a network of interconnected brokering nodes. The properties of such a network and

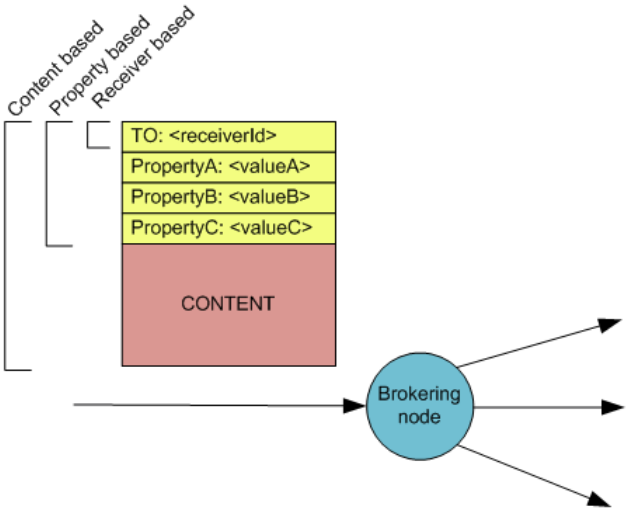

Fig. 12: Information used for message routing

the routing algorithms used for message distribution directly affect the overall system efficiency, scalability and robustness.

In typical scenarios, message-oriented middleware is used for sending messages which represent asynchronous service requests or generated events. However, rapid evolution of the SOA paradigm has resulted in additional demands placed on MOM, including efficient batch data transfer in the many-receivers model. Publish-subscribe delivery can be seen as application-level implementation of multicast communication. Apart from local network settings administered by a single authority, such services are unavailable in the network layer. Currently supported QoS assurances, such as message ordering and flow control mechanisms, already enable MOM for batch data delivery. However, in order to achieve a high level of effectiveness and scalability, the infrastructure has to be extended with additional mechanisms. One of them is multipathing, where data is transferred not only along a distribution tree constructed by the messaging middleware but also via additional paths. Extending the basic set of routes results in accelerated data transfer.

Another MOM enhancement from which data delivery solutions would benefit is caching. Assuming than a certain amount of disk storage is available at each brokering node, specific data access patterns can be identified. In such a case data can be cached in nodes close to the clients which most often request it. In currently available middleware solutions delivery buffers at each brokering node can be considered a basic caching mechanism. However, in order to provide suitable QoS, this approach is not satisfactory.

Finally, certain applications demand data transformation services. Prior to data consumption, clients have to preprocess data that is being sent. Assuming

Capability	Description
Transactions	Defined per communication session (in single-receiver or many-receivers model). Ensures that messages are sent and received in an all-or-nothing manner.
Message ordering	Specifies message ordering requirements. Some commonly available options include preserving the order of messages sent by a single user and delivered to every receiving site. Other possibilities include total ordering of messages or less restrictive casual order.
Persistency	Transport-level message persistency which enable communication even when some receivers are not currently available. Depending on infrastructure configuration, in-memory or data storage persistency can be used.
Flow control	Ensures that the rate of the message stream produced by the sender doesn't exceed the processing capabilities of the receivers.
Delivery and dispatch policies	Policies which specify what measures should be taken in case of delivery or dispatch failures.

Table 2: Message-oriented middleware QoS summary

that not all clients are able to perform appropriate transformations due to insufficient hardware resources or available software services, it appears appropriate to embed such capabilities in the data delivery infrastructure.

10 Summary

SOA application deployment and execution should take into account end-user requirements expressed as nonfunctional properties. Most of these requirements refer to QoS, reliability or security aspects which must be enforced during runtime. As many SOA applications are deployed as collections of communicating services, their governance should be QoS-aware and mindful of integration aspects. This explains the increasing significance of various extensions of the ESB technology, seen as a cornerstone of modern SOA execution infrastructures. The place and role of this technology is very well defined by the S3 model. Adaptive ESB addresses many governance aspects related to nonfunctional requirements. The proposed mechanisms and their building blocks create a consistent framework which can be further refined according to specific system needs.

The elements of the adaptive ESB framework, such as the monitoring system, exposition layer and policy engine are generic and can be applied to different layers of the S3 model. This is evidenced by the SCA extension supporting adaptive activity and the proposed BEPL monitoring system. The presented study underline the fundamental role of the SOA execution environment and application

monitoring as a starting point for SOA governance in general. Adaptive systems enable mature management of dynamic software execution environments.

Another key trend in the emergence of dynamic software systems. A dynamic system can be modified at runtime, without having to be restarted. This crucial feature is supported by the OSGi technology as far as Java-based systems are concerned. The OSGi model, often described as SOA for JVM, matches the concept of adaptive ESB or SCA. Dynamic software behavior can be further extended by AOP programming. This technology is very useful for instrumenting complex software systems such as ESB.

The presented overview can be summarized by stating that the proposed adaptive system concepts are well supported by modern software tools, opening a wide area for deployment and uptake by the software industry.

References

1. "The Forrester Wave: Enterprise Service Buses, Q1 2009," http://www.forrester.com/.
2. D. Chappell, T. Erl, M. Little, B. Loesgen, S. Roy, T. Rischbeck, and A. Simon, *Modern SOA Infrastructure: Technology, Design, and Governance.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 2010.
3. T. Erl, *SOA Design Patterns.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 2009.
4. T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005.
5. T. O. Alliance, "About the OSGi Service Platform," July 2004. [Online]. Available: http://www.osgi.org/documents/osgi_technology/osgi-sp-overview.pdf
6. *JSR 208: Java Business Integration (JBI).* [Online]. Available: http://jcp.org/en/jsr/detail?id=208
7. C. Hayman, "The Benefits of an Open Service Oriented Architecture in the Enterprise," 2005.
8. Taft and K. Darryl, "Can JBI 2 Succeed Where JBI 1 Did Not?" *eWeek*, May 2007. [Online]. Available: http://www.eweek.com/c/a/Application-Development/Can-JBI-2-Succeed-Where-JBI-1-Did-Not/
9. B. Hutchison, M.-T. Schmidt, D. Wolfson, and M. Stockton, "Soa programming model for implementing web services, part 4: An introduction to the ibm enterprise service bus," http://www.ibm.com/developerworks/library/ws-soa-progmodel4/ [Accessed 01/04/2009].
10. C. Lohman, L. Fortuin, and M. Wouters, "Designing a performance measurement system: A case study," *European Journal of Operational Research*, vol. 156, no. 2, pp. 267–286, July 2004. [Online]. Available: http://www.sciencedirect.com/science/article/B6VCT-48GP7GD-1/2/7bb13c9633ae1ea4b746043c758c2297
11. S. Amnajmongkol, J. Ang'ani, Y. Che, T. Fox, A. Lim, and M. Keen, "Business activity monitoring in websphere business monitor v6.1," IBM Redbooks, Tech. Rep., Jul. 2008.
12. *IEEE 802.1q: VLAN*, IEEE, http://www.ieee802.org/1/pages/802.1Q.html, 2005.
13. *SCA Service Component Architecture. Assembly Model Specification, version 1.00*, OASIS, May 2007.

14. "Apache Tuscany Home Page," http://tuscany.apache.org/, accessed on September 2009.
15. "Drools : Business Logic integration Platform," http://www.jboss.org/drools/, accessed on November 2009.
16. "JESS: the Rule Engine for the Java Platform," http://www.jessrules.com/, accessed on December 2008.
17. M. Burstein, J. Hobbs, O. Lassila, D. Mcdermott, S. Mcilraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara, "OWL-S: Semantic Markup for Web Services," Website, World Wide Web Consortium, November 2004. [Online]. Available: http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/
18. A. Ali, Z. Liang-Jie, E. Michael, A. Abdul, and C. Kishore, "Design an soa solution using a reference architecture, improve your development process using the soa solution stack," *IBM developerWorks*, 2007. [Online]. Available: http://www.ibm.com/developerworks/library/ar-archtemp/
19. T. W. M. Coalition, "The workflow management coalition," 2009, http://www.wfmc.org/.
20. W. M. Coalition, "Workflow management coalition, terminology & glossary," Workflow Management Coalition, Tech. Rep., 1999, http://www.wfmc.org/standards/docs/TC-1011_term_glossary_v3.pdf.
21. B. P. M. Initiative, "Business process management initiative," 1997, http://www.bpmi.org/.
22. D. T. and J. J, "Master thesis: Badanie wydajnoci systemw o architekturze," 2007.