The VLDB Journal (2007) 16:389–415
DOI 10.1007/s00778-007-0044-3

REGULAR PAPER

Service oriented architectures: approaches, technologies and research issues

Mike P. Papazoglou · Willem-Jan van den Heuvel

Received: 6 June 2005 / Accepted: 1 August 2005 / Published online: 3 March 2007
© Springer-Verlag 2007

Abstract Service-oriented architectures (SOA) is an emerging approach that addresses the requirements of loosely coupled, standards-based, and protocol-independent distributed computing. Typically business operations running in an SOA comprise a number of invocations of these different components, often in an event-driven or asynchronous fashion that reflects the underlying business process needs. To build an SOA a highly distributable communications and integration backbone is required. This functionality is provided by the Enterprise Service Bus (ESB) that is an integration platform that utilizes Web services standards to support a wide variety of communications patterns over multiple transport protocols and deliver value-added capabilities for SOA applications. This paper reviews technologies and approaches that unify the principles and concepts of SOA with those of event-based programing. The paper also focuses on the ESB and describes a range of functions that are designed to offer a manageable, standards-based SOA backbone that extends middleware functionality throughout by connecting heterogeneous components and systems and offers integration services. Finally, the paper proposes an approach to extend the conventional SOA to cater for essential ESB requirements that include capabilities such as service orchestration, "intelligent" routing, provisioning, integrity and security of message as well as service management. The layers in this extended SOA, in short xSOA, are used to classify research issues and current research activities.

M. P. Papazoglou (✉) · W.-J. van den Heuvel
INFOLAB, Tilburg University, PO Box 90500,
5000 LE, Tilburg, The Netherlands
e-mail: mikep@uvt.nl

Keywords Service oriented architecture · Asynchronous and event-driven processing · Application and service integration · Enterprise bus · Web services

1 Introduction

Modern enterprises need to respond effectively and quickly to opportunities in today's ever more competitive and global markets. To accommodate business agility, enterprises are supposed to streamline (existing) business processes while exposing the various packaged and home-grown applications found spread throughout the enterprise in a highly standardized manner. A contemporary approach for addressing these critical issues is embodied by (Web) *services* that can be easily assembled to form a collection of autonomous and loosely coupled business processes.

The emergence of Web services developments and standards in support of automated business integration has driven major technological advances in the integration software space, most notably, the service-oriented architecture (SOA) ([15,18]). The purpose of this architecture is to address the requirements of loosely coupled, standards-based, and protocol-independent distributed computing, mapping enterprise information systems (EIS) appropriately to the overall business process flow.

In an SOA, software resources are packaged as "services", which are well defined, self-contained modules that provide standard business functionality and are independent of the state or context of other services. Services are described in a standard definition language, have a published interface, and communicate with each

other requesting execution of their operations in order to collectively support a common business task or process [41]. Services that utilize Web services standards, e.g., Web Services Description Language (WSDL), Simple Object Access Protocol (SOAP), and Universal Description, Discovery and Integration registry (UDDI), are the most popular type of services available today.

An SOA is designed to allow developers to overcome many distributed enterprise computing challenges including application integration, transaction management, security policies, while allowing multiple platforms and protocols and leveraging numerous access devices and legacy systems [1]. The driving goal of SOA is to eliminate these barriers so that applications integrate and run seamlessly. In this way an SOA can deliver the flexibility and agility that business users require, defining coarse grained services, which may be aggregated and reused to facilitate ongoing and changing needs of business, as the key building blocks of enterprises.

In contrast to conventional software architectures primarily delineating the organization of a system in its (sub)systems and their interrelationships, the SOA captures a logical way of designing a software system to provide services to either end-user applications or other services distributed in a network through published and discoverable interfaces. SOA is focused on creating a design style, technology, and process framework that will allow enterprises to develop, interconnect, and maintain enterprise applications and services efficiently and cost-effectively. While this objective is definitely not new [66], SOA seeks to eclipse previous efforts such as modular programing, code reuse, and object-oriented software development techniques.

The SOA as a design philosophy is independent of any specific technology, e.g., Web-services or J2EE enterprise beans. This is achieved by limiting the number of implementation restrictions to the level of the service interface. SOA requires that functions, or services, are defined by a description language (WSDL [30] in the case of Web services) and have interfaces that perform useful business processes. The fundamental intent of a service in an SOA is to represent a reusable unit of business-complete work. A service in SOA is an exposed piece of functionality with three essential properties. Firstly, an SOA-based service is self-contained, i.e., the service maintains its own state. Secondly, services are platform independent, implying that the interface contract to the service is limited to platform independent assertions. Lastly, the SOA assumes that services can be dynamically located, invoked and (re-)combined.

Logically, a service in an SOA is a bound pair of a service interface and a service implementation. Service interface defines the identity of a service and its invocation logistics. Service implementation implements the work that the service is designated to do. Because interfaces are platform independent, a client from any communication device using any computational platform, operating system and any programing language can use the service. These two facets of the service are designed and maintained as distinct items, though their existence is highly interrelated.

An SOA provides a flexible architecture that unifies business processes by modularizing large applications into services. A client from any device, using any operating system, in any programing language, can access an SOA service to create a new business process. An SOA creates a collection of services that can communicate with each other using service interfaces to pass messages from one service to another, or coordinating an activity between one or more services.

Services used in composite applications may be brand new service implementations, they may be fragments of old applications that were adapted and wrapped, or they may be combinations of the above. Often the designers for the client of the service do not have direct access to the service implementation, other than indirectly through its interface. External Internet-based service providers and SOA packaged applications simply offer the interfaces without revealing the inner workings of their environment. Thus, with SOA, an enterprise can create, deploy and integrate multiple services and choreograph new business functions by combining new and existing application assets into a logical flow. Accordingly, for well defined and semantically unambiguous applications an SOA can serve as an enabler of just-in-time integration and interoperability of legacy applications; a key consideration for enterprises that are seeking to deploy demand driven computing environments.

Services in an SOA exhibit the following main characteristics [47]:

- All functions in an SOA are defined as services [7]. This includes pure business functions, business transactions composed of lower-level functions, and system service functions as well.
- All services are autonomous. Their operation is perceived as opaque by external components. Service opaqueness guarantees that external components neither know nor care how services perform their function, they merely anticipate that they return the expected result. The implementation and execution space of the application providing the desired functionality is encapsulated behind the service interface.

– In the most general sense, the interfaces are invocable. This implies that it is irrelevant whether services are local or remote, the interconnect scheme or protocol to effect the invocation, nor which infrastructure components are required to establish the connection.

The SOA's loose-coupling principle—especially the clean separation of service interfaces from internal implementations—to guide planning, development, integration, and management of their network application platforms make them indispensable for enterprise-wide and cross-enterprise applications [7].

Web services seem to become the preferred implementation technology for realizing the SOA promise of maximum service sharing, reuse, and interoperability [56]. Web services and SOA reduce complexity of enterprise application eco-systems by encapsulation and minimizing the requirements for shared understanding by defining service interfaces in an unambiguous and transparent manner. Also Web services enable just-in-time integration and interoperability of legacy applications. Based on open and pervasive standards, Web services seem poised for success, as these are only built on top of existing, ubiquitous infrastructure like HTTP, SOAP, and XML.

In this article, we survey the underpinnings of SOA, assessing methodologies and technologies that serve as the enabling springboard for business integration projects and deliver a flexible and adaptable environment. This survey is unique in that it unifies the principles, concepts and developments in the area of application integration, middleware, and integration brokers, SOA, event-driven computing, and adapters, and explains how they operate as part of an emerging distributed computing technology named the Enterprise Service Bus (ESB). Moreover, this paper develops and explores an extension to conventional SOAs, entitled the eXtended SOA (xSOA). xSOA is an attempt to streamline, group together, and logically structure the functional requirements of complex applications that make use of the service-oriented computing paradigm. xSOA serves as the reference for reviewing available technologies, and assessing open research issues.

2 Service roles in SOA

The SOAs and Web services solutions support two key roles: a service requestor (client) and service provider, which communicate via service requests. A role thus reflects a type of a participant in an SOA ([18,55]).

Service requests are messages formatted according to the Simple Object Access Protocol (SOAP) [16]. SOAP entails a light-weighted protocol allowing RPC-like calls over the Internet using a variety of transport protocols including HTTP, HTTP/S and SMTP. In principle, SOAP messages may be conveyed using any protocol as long as a binding is defined. The SOAP request is received by a run-time service (a SOAP "listener") that accepts the SOAP message, extracts the XML message body, transforms the XML message into a native protocol, and delegates the request to the actual business process within an enterprise.

Requested operations of Web services are implemented using one or more Web service components [103]. Web service components may be hosted within a Web services container [36], serving as an interface between business services and low-level, infrastructure services. In particular, Web service containers are similar to J2EE containers [2] providing facilities such as location, routing, service invocation, and management. In particular, a service container is the physical manifestation of the abstract service endpoint, and provides the implementation of the service interface. In addition, service containers provide facilities for lifecycle management such as start up, shutdown, and resource cleanup. A service container can host multiple services, even if they are not part of the same distributed process. Thread pooling allows multiple instances of a service to be attached to multiple listeners within a single container [27]. Finally, the response that the provider sends back to the client takes again the form of a SOAP envelope carrying an XML message.

SOAP is by nature a platform-neutral and vendor-neutral standard. These characteristics allow for a loosely coupled relationship between requester and provider, which is especially important over the Internet where two parties may reside in different organizations or enterprises. However, SOA does not require the usage of SOAP. Prior to SOAP, for example, some companies used IBM's WebSphere MQ to exchange XML documents between them [99]. While this type of infrastructure clearly does not support Web services because they communicate using SOAP, they are another example of service invocation in an SOA. Currently WebSphere MQ is, of course, equipped with direct support for SOAP.

While SOA services are visible to the service client, their underlying Web components are transparent. The service consumer does not have to be concerned with the implementation of the service, as long as it supports the required functionality, while offering the desired quality of service. For the service provider, the design of components, their service exposure and management reflect key architecture and design decisions that enable

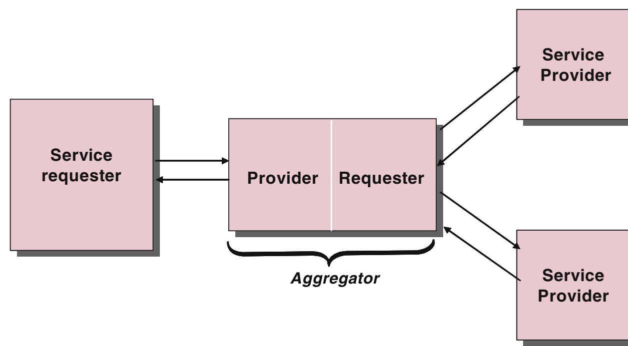

Fig. 1 The role of service aggregator

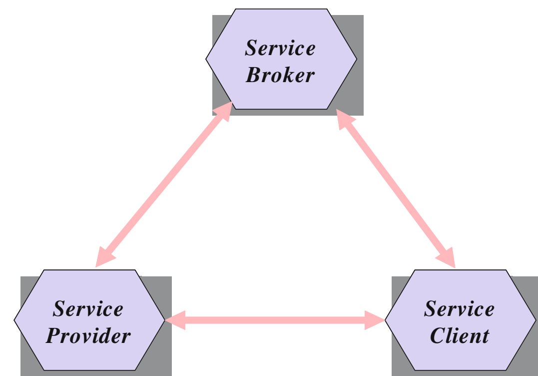

Fig. 2 Service brokering

services in SOA. The provider view offers a perspective on how to design the realization of the component that offers the services; its architectural decisions and designs.

The process of a service requester having to directly interact with a service provider exposes service requesters to the potential complexity of discovering, exploring, negotiating, and reserving services between different service providers. An alternative approach is for an organization to provide this combined functionality directly to the service requester. This service role combines the role of service requester and provider, and is labeled as a service aggregator [73]. The service aggregator thus performs a dual role. First, it acts as an application service provider as it offers a complete "service" solution by creating composite, higher-level services, which it provides to the service client. Service aggregators can accomplish this composition using specialized composition languages like BPEL [4] and BPML [5]. Second, it acts as a service requester as it may need to request and reserve services from other service providers. This process is shown in Fig. 1.

While service aggregation may offer direct benefits to the requester, it is a form of service brokering that offers a convenience function—all the required services are grouped "under one roof". However, an important question that needs to be addressed is how does a service requester determine which one out of a number of potential application service providers, should be selected for their service offerings? The service requester could retain the right to select an application service provider based on those that can be discovered from a registry service, such as the UDDI [94]. SOA technologies, such as UDDI, and security and privacy standards such as SAML [39] and WS-Trust [3], introduce another role which addresses these issues, called a service broker [31].

Service brokers are trusted parties that force service providers to adhere to information practices that comply with privacy laws and regulations, or in the absence of such laws, industry best practices. In this way, broker-sanctioned service providers are guaranteed to offer services that are in compliance with local regulations and create a more trusted relationship with customers and partners. A service broker maintains an index of available service providers. The service broker is able to "add value" to its registry of application service providers by providing additional information about their services. This may include differences about the reliability, trustworthiness, the quality of the service, service level agreements, and possible compensation routes to name a few.

Figure 2 shows an SOA where a service broker serves as an intermediary that is interposed between service requesters and service providers. Figure 2 falls under this category with the service registry (UDDI operator) being a specialized instance of a service broker. Under this configuration, the UDDI registry serves as a broker where the service providers publish the definitions of the services they offer using WSDL and where the service requestors find information about the services available.

3 Enterprise service bus

Essentially, Web services denote an important technology for implementing SOAs; however, other more conventional programing languages or middleware platforms may be adopted as well [91]. In particular, all technologies that directly implement service interfaces with WSDL, and communicate with XML messages, can be involved in SOA. As indicated earlier, other technologies such as, for instance, established middleware technologies like J2EE, CORBA, and IBM's WebSphere MQ, can now also participate in an SOA, using new features that work with WSDL.

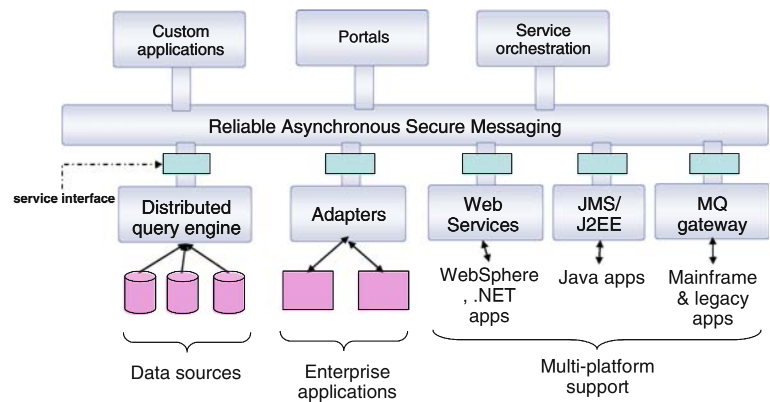

Fig. 3 Enterprise service bus connecting diverse applications and technologies

Fundamentally, there are only two options for solving technology and information model mismatches:

1. Build the client module to conform exactly to the characteristics of every server module that it will invoke.
2. Insert a layer of communication and integration logic between the client and server modules.

The first approach requires to "develop an interface" for each connection, resulting in a point-to-point topology. This topology has long been known to be hard to manage and maintain as it introduces a tighter form of coupling to harmonize transport protocols, document formats, interaction styles, etc. [61]. This makes it harder to change either of the two systems involved in an interchange without impacting other systems. In addition, point-to-point integration is not scalable and very complex as the number of integration points increases as the number of systems increases and can quickly become unmanageable. Hence, the broad use of Enterprise Application Integration middleware supporting a variety of hub-and-spoke integration patterns [74]. This leaves the second approach as the most viable alternative.

The second approach introduces an integration layer that must support interoperability among, and coexist with deployed infrastructure and applications. The requirements to provide an appropriately capable and manageable integration infrastructure for Web services and SOA are coalescing into the concept of the ESB ([27,82]). The ESB exhibits two prominent features [52]. Firstly, it promotes loose coupling of the systems taking part in an integration. Secondly, the ESB can break up the integration logic into distinct easily manageable pieces.

The ESB is an open, standards-based message bus designed to enable the implementation, deployment, and management of SOA-based solutions with a focus on assembling, deploying, and managing distributed SOA. The ESB provides the distributed processing, standards-based integration, and enterprise-class backbone required by the extended enterprise [52]. The ESB is designed to provide interoperability between large-grained applications and other components via standards-based adapters and interfaces. The bus functions as both transport and transformation facilitator to allow distribution of these services over disparate systems and computing environments.

Conceptually, the ESB has evolved from the store-and-forward mechanism found in middleware products, e.g., Message Oriented Middleware, and combines conventional EAI technologies with Web services, XSLT [98], orchestration, and choreography technologies, e.g., BPEL, WS-CDL, and ebXML BPSS. Physically, an ESB provides an implementation backbone for an SOA. It establishes proper control of messaging as well as applies the needs of security, policy, reliability, and accounting, in an SOA architecture. The ESB, is responsible for the proper control, flow, and translations of all messages between services, using any number of possible messaging protocols. An ESB pulls together applications and discrete integration components to create assemblies of services to form composite business processes, which in turn automate business functions in an enterprise.

Figure 3 depicts a simplified architecture of an ESB that integrates a J2EE application using JMS, a .NET application using a C# client, an MQ application that interfaces with legacy applications, as well as external applications and data sources using Web services. An ESB, as portrayed in Fig. 3, enables the more efficient value-added integration of a number of different application components, by positioning them behind a service-oriented facade and by applying Web services technology. In this figure, a distributed query engine, which is normally based on XQuery [14] or SQL, enables the creation of data services to abstract the complexity

of underlying data sources. A portal in Fig. 3 is a user-facing aggregation point of a variety resources represented as services, e.g., retail, divisional, corporate employee, and business partner portals.

Endpoints in the ESB, depicted in Fig. 3, provide abstraction of physical destination and connection information (like TCP/IP hostname and port number) transcending plumbing level integration capabilities of traditional, tightly coupled, distributed software components. Endpoints allow services to communicate using logical connection names, which an ESB will map to actual physical network destinations at runtime. This destination independence offers the services that are connected to the ESB, the ability to be upgraded, moved, or replaced without having to modify code and disrupt existing ESB applications. For instance, an existing ESB invoicing service could be easily upgraded by a new service, without disrupting the execution of other applications. Additionally, duplicate processes can be set up to handle fail-over if a service is not available. Endpoints also provide the asynchronous and highly reliable communication between service containers. The endpoints can be configured to use several levels of quality of service, which guarantee communication despite network failures and outages [27].

To successfully build and deploy a distributed SOA, there are four primary aspects that need to be addressed:

1. *Service enablement* Each discrete application needs to be exposed as a service.
2. *Service orchestration* Distributed services need to be configured and orchestrated in a unified and clearly defined distributed process.
3. *Deployment* Emphasis should be shifted from test to the production environment, addressing security, reliability, and scalability concerns.
4. *Management* Services must be audited, maintained and reconfigured. The latter requirements requires that corresponding changes in processes must be made without rewriting the services or underlying application.

Services can be programed using application development tools (like Microsoft .NET, Borland JBuilder, or BEA WebLogic Workshop), which allow new or existing distributed applications to be exposed as Web services. Technologies like J2EE Connector Architecture (JCA) may also be used to create services by integrating packaged applications (like ERP systems), which would then be exposed as services.

To achieve its operational objectives, the ESB draws from traditional EAI broker functionality in that it provides integration services such as connectivity and routing of messages based on business rules, data transformation, and adapters to applications [28]. These capabilities are themselves SOA-based in that they are spread out across the bus in a highly distributed fashion and hosted in separately deployable service containers. This is crucial difference from traditional integration brokers, which are usually highly centralized and monolithic in nature [74]. The SOA approach allows for the selective deployment of integration broker functionality exactly where it is needed with no additional over-bloating. The distributed nature of the ESB container model allows individual event-driven services to plugged into the ESB backbone on an as needed basis, be highly decentralized and work together in a highly distributed fashion, while they are scaled independently from one another. This is illustrated in Fig. 3 where applications running on different platforms are abstractly decoupled from each other, and can be connected together through the bus as logical endpoints that are exposed as event-driven services.

3.1 Event-driven SOA

In the enterprise context, business events, e.g., a customer order, the arrival of a shipment at a loading dock, or the payment of a bill, may affect the normal course of a business process at any point in time [64]. This implies that business processes cannot be designed a priori assuming that events are predetermined following a particular flow, but must be defined dynamically, driven by incoming, parallel and a synchronous event flows. Supporting enterprise applications then must communicate using an event-driven SOA ([27,89]). An event-driven SOA thus denotes an architectural approach on how enterprises could implement an SOA, respecting the highly volatile nature of business events. An ESB requires that applications and event-driven services are tied together in the context of an SOA in a loosely coupled fashion. This allows them to operate independently from each other while still providing value to a broader business function [28].

In an ESB-enabled event-driven SOA, applications and services are treated as abstract service endpoints, which can readily respond to asynchronous events [28]. The event-driven SOA provides a means of abstracting away from the details of underlying service connectivity and protocols.

Services in this SOA variant are not required to understand protocol implementations or have any knowledge on routing of messages to other services. An event source typically sends messages through some middleware integration mechanism like an ESB, and then the

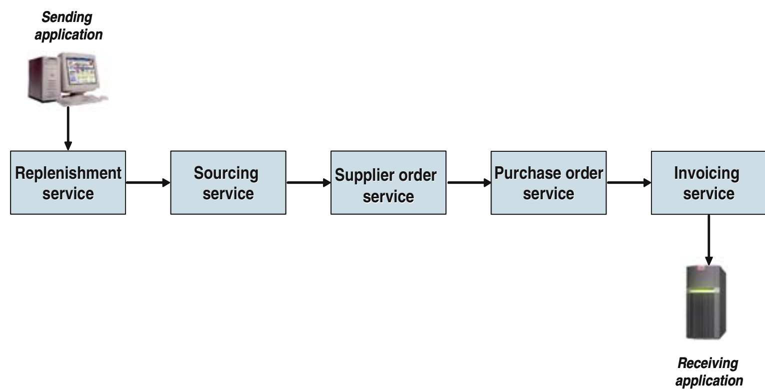

Fig. 4 Simplified distributed procurement process

middleware publishes the messages to the services that have subscribed to the events. The event itself encapsulates an activity, constituting a complete description of a specific action. To achieve its functionality, the ESB supports both the established Web services technologies, including, SOAP, WSDL, and BPEL, as well as emerging standards such as WS-ReliableMessaging [49] and WS-Notification [44].

As we already explained earlier in the previous section, in a brokered SOA (see Fig. 2) the only dependency between the provider and client of a service is the service contract (described in WSDL), which the third-party service registry advertises. The dependency between the service provider and the service client is a run-time dependency, not a compile-time dependency. The client obtains and uses all the information it needs about the service at run-time. The service interfaces are discovered dynamically, and messages are constructed dynamically. The service consumer does not know the format of the request message or response message or the location of the service until it needs a particular service.

Service contracts and other associated meta-data, including meta-data about policies and agreements [33], lay the groundwork for enterprise SOAs that involve many clients operating with a complex, heterogeneous application infrastructure. However, many of today's SOA implementations are not that elaborate. In many cases, when small or medium enterprises implement SOA, neither service interfaces in WSDL nor UDDI look-ups tend to be available. This is often either due to the fact that the SOA in place provides for limited functionality or because sufficient security arrangements are not yet in place. In these cases, an event-driven SOA provides a more lightweight, straightforward set of technologies to build and maintain the service abstraction for client applications [13].

To achieve a more lightweight arrangement an event-driven SOA requires that two participants in an event (server and client) be fully decoupled [13], not just loosely coupled. With fully decoupled exchanges the two participants in an event need not have any knowledge about each other, before engaging in some business transaction. In this case, there is no need for a service (WSDL) contract that explicates the behavior of a server to the client. The only relationship is indirect, through the ESB, to which clients and servers are subscribed as subscribers and publishers of events. Despite the notion of decoupling in event-driven SOA, recipients of events require meta-data about those events. The publishers of the events often organize them on the basis of some (topical) taxonomy, which is a form of meta-data. Alternatively, meta-data is available about the event, including its size, format, etc. In contrast to service interfaces, however, meta-data that is associated with events is generated on an ad hoc basis, instead of being static and predefined. In particular, ad hoc meta-data describe published events that consumers can subscribe to, the interfaces that service clients and providers exhibit as well as the messages they exchange, and even the agreed format and context of those meta-data, without falling into the formal service contracts themselves.

To effectively orchestrate the behavior of services in a distributed process, the ESB infrastructure includes a distributed processing framework and XML-based services. To exemplify these features, a simplified distributed procurement business process as shown in Fig. 4, will be configured and deployed using an ESB. The figure shows that when an automated inventory system triggers a replenishment signal, an automated procurement process flow is triggered and a series of logical steps need to be performed. First the sourcing service queries the enterprise's supplier reference database to determine the list of possible suppliers, who could be prioritized on the basis of existing contracts and supplier metrics. A supplier is then chosen based on some criterion and the purchase order is automatically generated in an ERP purchasing module and is sent the vendor of

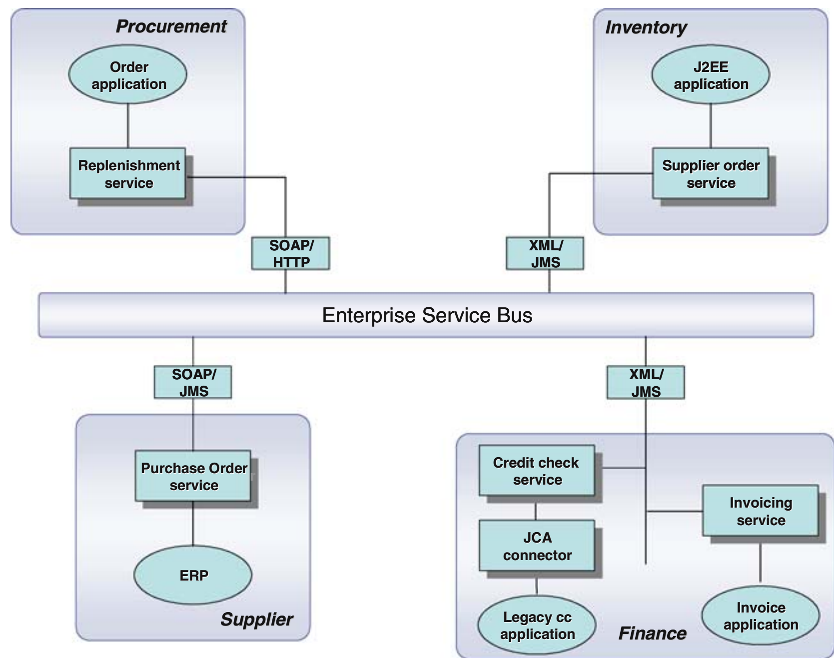

Fig. 5 Enterprise service bus connecting remote services

choice. Finally, this vendor uses an invoicing service to bill the customer.

The ESB distributed processing infrastructure is aware of applications and services and uses content-based routing facilities to make informed decisions about how to communicate with them. The services that are part of the distributed procurement business process depicted in Fig. 4 can be seen in use in Fig. 5. For this example, the inventory is assumed to be out of stock, and the replenishment message is routed to a supplier order service. Although this figure shows only a single supplier order service as part of the inventory, in reality a plethora of supplier services may exist. The supplier order service, which executes a remote Web service at a chosen supplier to fulfil the order, generates its output in an XML message format that is not understood by the purchase order service. To avoid heterogeneity problems, the message from the supplier order service leverages the ESB's transformation service to convert the XML into a format that is acceptable by the purchase order service. Fig. 5 also shows that JCA is used within the ESB to allow legacy applications, such as credit check service, to be placed onto the ESB through JCA resource adapters.

Once services that are part of the distributed procurement business process depicted in Fig. 4 have been chained together, it is essential to provide a way to manage and reconfigure them to react to changes in business processes. Ideally, this could be achieved through a sophisticated graphical business process management tool that can be used to configure, deploy, and manage services and endpoints. This allows the free movement and reconfiguration of services without requiring rewriting or modifying the services themselves.

3.2 Key capabilities

In order to implement an SOA, both applications and infrastructure must support SOA principles (see Sect. 1). Enabling an application for SOA involves the creation of service interfaces to existing or new functions, either directly or through the use of adaptors. Enabling the infrastructure, at the most basic level, involves provision of the capabilities to route and deliver service requests to the correct service provider. However, it is also vital that the infrastructure supports safe substitution of one service implementation by another, without any effect to the clients of that service. This requires not only that the service interfaces be specified according to SOA principles, but also that the infrastructure allows client code to invoke services irrespectively of the service location and the communication protocol involved. Such service routing and substitution are among the many capabilities of the ESB. Additional capabilities can be found in the following list that describes detailed functional requirements for an ESB.

We consider the following ESB capabilities list to be necessary to support the functions of a useful and meaningful ESB. Some of the ESB functional requirements described in the list below have also been discussed by other authors such as ([23,27,47,79]).

- *Leveraging existing assets* Legacy applications are technically obsolete mission critical elements of an organization's infrastructure—as they form the core of larger enterprise's business processes—but are too frail to modify and too important to discard and thus must be reused. Strategically, the objective is to build a new architecture that will yield all the value hoped for, but tactically, legacy assets must be leveraged and integrated with modern technologies and applications.
- *Service communication capabilities* A critical ability of the ESB is to route service interactions through a variety of protocols, and to transform from one protocol to another where necessary. Another important aspect of an ESB implementation is the capacity to support service messaging models consistent with the SOA interfaces, as well as the ability of transmitting the required interaction context, such as security, transaction, or message correlation information.
- *Dynamic connectivity capabilities* Dynamic connectivity pertains to the ability to connect to Web services dynamically without using a separate static API or proxy for each service. Most enterprise applications today operate on a static connectivity mode, requiring some static piece of code for each service. Dynamic service connectivity is the key capability for a successful ESB implementation. The dynamic connectivity API is the same regardless of the service implementation protocol (Web services, JMS, EJB/RMI, etc.).
- *Topic/content-based routing capabilities* The ESB should be equipped with routing mechanisms to facilitate not only topic-based routing but also, more sophisticated, content-based routing. Topic-based routing assumes that messages can be grouped into fixed, topical classes, so that subscribers can explicate interest in a topic and as a consequence receive messages associated with that topic [71]. Content-based routing on the other hand allows subscriptions on constraints of actual properties (attributes) of business events. The content of the message thus determines their routing to different endpoints in the ESB infrastructure. For example, if a manufacturer provides a wide variety of products to its customers, only some of which are made in-house, depending on the product ordered it might be necessary to direct the message to an external supplier, or route it to be processed by an internal warehouse fulfilment service. In a typical application, a message is routed by opening it up and applying a set of rules to its content to determine the parties interested in its content. Content-based routing is often dependant on the message constructed in XML, and will usually be built on top of Message Oriented Middleware, or JMS based messaging. Such ESB capabilities could be based on emerging standard efforts such as WS-Notification.

  WS-Notification defines a general, topic-based Web service system for publish and subscribe interactions, which relies on the WS-Resource framework [43]. WS-Notification [44] is a family of related specifications that define a standard Web services approach to notification using a topic-based publish/subscribe pattern. The WS-Notification specification defines standard message exchanges to be implemented by service providers that wish to participate in notifications and standard message exchanges—allowing publication of messages from entities that are not themselves service providers. It also allows expressing operational requirements expected of service providers and requesters that participate in notifications. WS-Notification allows notification messages to be attached to WSDL PortTypes. The current WS-Notification specification provides support for both brokered as well as peer-to-peer publish/subscribe.
- *Endpoint discovery with multiple quality of service capabilities* The ESB should support the fundamental SOA need to discover, locate, and bind to services. Increasingly these capabilities will be based around Web services standards such as WSDL, SOAP, UDDI, and WS-PolicyFramework. As many network endpoints can implement the same service contract, the ESB should support service interactions with different values to the business. Several scenarios make it desirable for the client to select the best endpoint at run-time, rather than hard-coding endpoints at build time. The ESB should therefore be capable of supporting various qualities of service. Clients can query a Web service, such as an organizational UDDI service, to discover the best instance with which to interact based on QoS properties. Ideally, these capabilities should be controlled by declarative policies associated with the services involved using a policy standard such as the WS-PolicyFramework [9].
- *Integration capabilities* To support SOA in a heterogeneous environment, the ESB needs to integrate with a variety of systems that do not directly support service-style interactions. These may include legacy systems, packaged applications, COTS components, etc. When assessing the integration requirements for ESB, several types or "styles" of integration must be considered. Due their importance ESB integration styles are discussed in some detail later in this article.

- *Transformation capabilities* The various components hooked into the ESB have their own expectations of messaging models and data formats, and these might differ from other components. A major source of value in an ESB is that it shields any individual component from any knowledge of the implementation details of any other component. The ESB transformation services make it possible to ensure that messages and data received by any component is in the format it expects, thereby removing the burden to make changes. The ESB plays a major role in transformations between different, heterogenous data and messaging models, whether between legacy data formats (e.g., a COBOL/VSAM application, running on an OS/390 host) and XML, between basic XML formats, and Web services messages, or between different XML formats (e.g., transforming an industry standard XML message to a proprietary or custom XML format).
- *Reliable messaging capabilities* Reliable messaging refers to the ability to queue service request messages and ensure guaranteed delivery of these messages to the destination. It also includes the ability to respond, if necessary, back to the requestor with response messages. Reliable messaging supports asynchronous store-and-forward delivery as well as guaranteed delivery capabilities. Primarily used for handling events, this capability is crucial for responding to clients in an asynchronous manner, and for a successful ESB implementation.
- *Security capabilities* Generically handling and enforcing security is a key success factor for ESB implementations. The ESB needs to both provide a security model to service consumers and integrate with the (potentially varied) security models of service providers. Both point-to-point (e.g., SSL encryption) and end-to-end security capabilities will be required. These end-to-end security capabilities include federated authentication, which intercepts service requests and adds the appropriate username and credentials; validation of each service request and authorization to make sure that the sender has the appropriate privilege to access the service; and, lastly, encryption/decryption of XML content at the element level for both message requests and responses. To address these intricate security requirements trust models, WS-Security [8] and other security related standards have been developed.
- *Long running process and transaction capabilities* Service-orientation, as opposed to distributed object architectures such as .NET or J2EE, make it possible to more closely reflect real-world processes and relationships. It is believed that SOA represents a much more natural way to model and build software that solves real-world business processing needs [7]. Accordingly, the ESB should provide the ability to support business processes and long running services—services that tend to run for long duration, exchanging message (conversation) as they progress. Typical examples are an online reservation system, which interacts with the user as well as various service providers (airline ticketing, car reservation, hotel reservation, etc.).

  In addition, it is of vital importance that the ESB provides certain transactional guarantees. More specifically, the ESB needs to be able to provide a means for various applications to interact and message with each other and to recover should some form of technical or process failure occur. The challenge at hand is to ensure that complex transactions are handled in a highly reliable manner and if failure should occur, transactions should be capable of rolling back processing to the original, pre-request state.
- *Management and monitoring capabilities* In an SOA environment, applications cross system (and even organizational) boundaries, they overlap, and they can change over time. Managing these applications is a serious challenge [6]. Examples include dynamic load balancing, fail-over when primary systems go down, and achieving topological or geographic affinity between the client and the service instance, and so on. Effective systems and application management in an ESB require a management framework that is consistent across an increasingly heterogeneous set of participating component systems, while supporting complex aggregate (cross-component) management use cases, like dynamic resource provisioning and demand-based routing, service-level agreement enforcement in conjunction with policy-based behavior. The latter implies the ability to select service providers dynamically based on the quality of service they offer compared to the business value of individual transactions.

  An additional requirement for a successful ESB implementation is the ability to monitor the health, capacity, and performance of services. Monitoring is the ability to track service activities that take place via the bus and accommodate visibility into various metrics and statistics. Of particular significance is the ability to be able to spot problems and exceptions in the business processes and move toward resolving them as soon as they occur. Process monitoring capabilities are currently provided by toolsets in platforms for developing, deploying and managing service applications, such as, for instance, WebLogic Workshop.

- *Scalability capabilities* With a widely distributed SOA, there will be the need to scale some of the services or the entire infrastructure to meet integration demands. For example, transformation services are typically very resource intensive and may require multiple instances across two or more computing nodes. At the same time, it is necessary to create an infrastructure that can support the large nodes present in a global service network. The loose coupled nature of an SOA requires that the ESB uses a decentralized model to provide a cost effective solution that promotes flexibility in scaling any aspect of the integration network. A decentralized architecture enables independent scalability of individual services as well as the communications infrastructure itself.

As ESB integration capabilities in the above list are central in understanding the material that follows and a key element of the ESB when performing service-oriented integration, we shall consider them in some detail in the remainder of this section.

3.3 Integration solutions

ESBs employ a service-oriented integration solution that leverages among other issues open standards, loose coupling, and the dynamic description and discovery capabilities of Web services to reduce the complexity, cost, and risk of integration. Other salient characteristics of the ESB architectural integration style are that it is technology agnostic and can reuse functionality in existing applications to support new application development. There is a series of important technical requirements that need to be addressed by a service-oriented integration solution ([47,52]). These include:

Integration at the presentation-tier Integration at the presentation-tier is concerned with how the complete set of applications and services a given user accesses are fabricating a highly distributed yet unified portal framework that provides a usable, efficient, uniform, and consistent presentation-tier. In this way, the ESB can provide one face to the users resulting in consistent user experience, with unified information delivery while allowing underlying applications remain distributed. Two complementary industry standards that are emerging in the portal space can assist with these efforts [57]:

1. JSR 168 This is an industry standard that defines a standard way to develop portlets. It allows portlets to be interoperable across portal vendors. For example, portlets developed for BEA WebLogic Portal can be interoperable with IBM Portal. This allows organizations to have a lower dependency on the portal product vendor.
2. WSRP (Web Service for Remote Portals) This is an industry standard that allows remote portlets to be developed and consumed in a standard manner and facilitates federated portals. WSRP combines the power of Web services and portal technologies and is fast becoming the major enabling technology for distributed portals in an enterprise.

JSR 168 complements WSRP by dealing with local rather than distributed portlets. A portal page may have certain local portlets which are JSR 168 compliant and some remote, distributed portlets that are executed in a remote container. With JSR 168 and WSRP maturing, the possibility of a true ESB federated portal can become a reality.

Application connectivity Application connectivity is an integration style concerned with all types of connectivity that the ESB integration layer must support. At the infrastructure level, this means concerns such as synchronous and asynchronous communications, routing, transformation, high speed distribution of data, and gateways and protocol converters. On the processing level, application connectivity also relates to the visualization of input and output, or sources and sinks.

Visualization signifies the fact that input is received and passed to applications in the ESB in a source-neutral way. Special purpose front-end device and protocol handlers should make that possible. For connectivity, an ESB can utilize J2EE components such as the Java Message Service for MOM connectivity, and J2EE Connector Architecture for connecting to application adapters. An ESB can also integrate easily with applications built with .NET, COM, C#, C++ and C. In addition, an ESB can integrate easily with any application that supports SOAP and Web services.

Application integration Application integration is concerned with building and evolving an integration backbone capability that enables fast assembly and disassembly of business software components. Application integration is an integral part of the assembly process that facilitates strategies which combine legacy applications, acquired packages, external application subscriptions and newly built components. The ESB should focus on a service-based application integration style that enables better-structured integration solutions that deliver:

- Applications comprised interchangeable parts that are designed to be adaptable to business and technology change.

– Evolutionary application portfolios that protect investment and can respond rapidly to new requirements and business processes
– Integration of various platform and component technologies.

Process integration Process integration is concerned with the development of automated processes that map to and provide solutions for business processes, integration of existing applications into business processes, and integrating processes with other processes. Process-level integration at the level of ESB generally includes the integration of business processes and applications within the enterprise (viz. EAI solutions). It may also involve the integration of whole processes, not simply individual services, from external sources, such as supply chain management or financial services that span multiple institutions (viz. e-Business integration solutions).

Data integration Information integration [76] is the process of providing a consistent access to all the data in the enterprise, by all the applications that require it, in whatever form they need it, without being restricted by the format, source, or location of the data. This requirement, when implemented, might involve adapters and transformation facilities, aggregation services to merge and reconcile disparate data, e.g., merging two customer profiles, and validation to ensure data consistency, e.g., minimum income should be equal to or exceed a certain threshold. Data should be transformed irrespectively of the formats under which they exist, the operating system that manages the data, and the location where the data are stored.

Integration design and development methodology One of the requirements for the application development environment must be that it takes into account all the styles and levels of integration that could be implemented within the enterprise, and provide for their development and deployment. To be truly robust, the development environment must rely on a methodology that clearly prescribes how services and components are designed and built in order to facilitate reuse, eliminate redundancy, and simplify integration, testing, deployment, and maintenance.

All of the styles of integration listed above will have some form of incarnation within any enterprise, even though in some cases they might be simplified or not clearly defined. It is important to note that all integration styles must be considered when embarking on an ESB implementation.

3.4 Enabling technologies

In this section, we will review the technological underpinnings of ESBs in some more detail. Fundamentally, ESBs fuse the following four types of technologies: integration brokers, application servers, business process management, and adapters.

3.4.1 Integration brokers

A prominent technology to interconnect disparate business applications, many of which are home-grown, ERP systems or legacy systems, constitutes integration brokers. Integration brokers come in many manifestations, ranging from early application-to-application brokers to more sophisticated broker topologies managing transactions, security, (resource) adapters and application protocols [22]. Some examples of commercially available integration brokers include IBM's WebSphere Integration Broker, PeopleSoft's AppConnect, and Sun ONE Integration Server.

Figure 6 presents a high-level view of the typical architecture for implementing integration broker. In particular, this figure illustrates the use of an integration broker to integrate functions and information from a variety of back-end EIS. To effectively illustrate the workings of the integration broker, we use the simplified distributed procurement process shown in Fig. 4. Figure 6 exemplifies that when an automated inventory system triggers a replenishment signal, an automated procurement process flow is triggered and first the enterprise's supplier reference database is queried to determine the list of possible suppliers, who could be prioritized on the basis of existing contracts and supplier metrics. A supplier is then chosen based and the purchase order is automatically generated in the ERP purchasing module and is sent to the vendor of choice.

The figure illustrates that the integration-broker is the system centrepiece. The integration-broker facilitates information movement between two or more resources (source and target applications, indicated by solid lines) in Fig. 6, and accounts for differences in application semantics and heterogeneous platforms. The various existing (or component) EIS, such as customer relationship management, ERP systems, transaction processing monitors, legacy systems, and so on, in this configuration are connected to the integration broker by means of resource adapters. This is indicated by the presence of dotted lines in Fig. 6. A resource adapter is used to provide access to a specific EIS. The purpose of the adapters is to provide a reliable insulation layer between application APIs and the messaging infrastructure. These adapters enable non-invasive application integration in a loosely coupled configuration.

Integration brokers are able to share information with a multitude of systems by using an asynchronous, event-driven mechanism thus they constitute an ideal

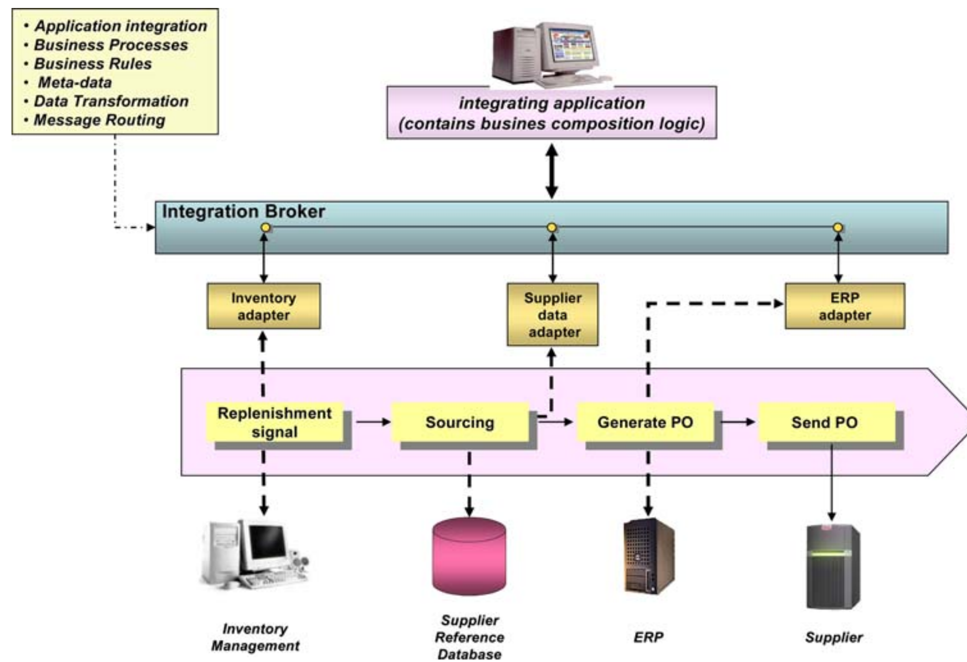

Fig. 6 Integration broker integrating disparate back-end systems

support framework for asynchronous business processes. Integration brokers, as realized in enterprise application integration suites, have historically been used for the integration of packaged applications via specific and often heavily customized adapters. Nowadays, this type of middleware is responsible for brokering messages exchanged between multiple applications, providing the ability to transform, store, and route messages, also the ability to apply business rules and respond to events.

The integration broker architecture presents several advantages as it tries to reduce the application integration effort by providing pre-built functionality common to many integration scenarios. The value proposition rests on reuse (in terms of middleware infrastructure and the application integration logic) across multiple applications and initiatives. Modern integration brokers incorporate integration functionality such as transformation facilities, process integration, business process management and trading partner management functionality, packaged adapters, and user-driven applications through front-end capabilities such as Java Server Pages (JSP).

3.4.2 Application servers

Application servers are widely used to develop and deploy back-end server logic. Application servers enable the separation of application (or business) logic and interface processing and also coordinate many resource connections. The most prominent features of application servers include secure transactional execution environment, load balancing, application-level clustering across multiple servers, failover management should one of these servers break down [61]. In addition, application servers provide application connectivity and thus access to data and functions associated with EIS applications, such as ERP, CRM, and legacy applications. While application servers are focused on supporting new application development, they do not natively support integration.

Application servers, such as SUN's J2EE [85], IBM's WebSphere [40], and Microsoft's .NET [62], typically provide Web connectivity for extending existing solutions and bring transaction processing mechanisms to the Web. In essence, an application server is simply a collection of services that support the development, runtime execution, and management of business logic for Web-enabled applications. The application server middleware enables the functions of handling transactions and extending back-end business data and applications to the Web. Application servers retrieve information from many existing enterprise systems and expose them through a single interface, typically a Web browser. This makes application servers ideal for portal-based EAI development. Unlike integration brokers, they do not integrate back-end systems directly.

Because application servers were created for Web-based transactions and application development and because of their ability to provide component-based integration to back-end applications, they are

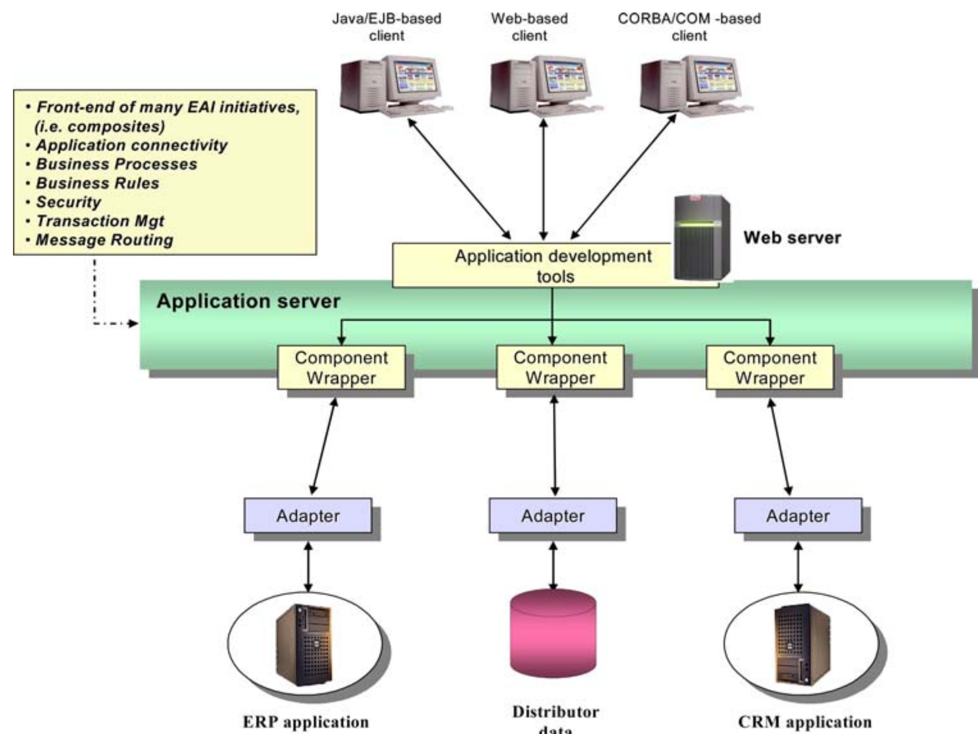

Fig. 7 Application server providing access to back-end systems

particularly useful as support framework for integrating business processes.

Figure 7 illustrates the use of an application server for a wholesale application that brings together ERP capabilities with sophisticated customer interfaces to open up new models of sales and distribution. The component wrappers in this figure facilitate point-integration of component systems, e.g., ERP, CRM, and distributor databases, as they introduce each of them to the application server. A component wrapper may be defined as a software layer that encapsulates legacy data and logic and defines its services in the wrapper API. The wrapper API mediates between calls from client application components to the underlying legacy code and data, by transforming incoming requests into a message format that is understandable to the internal code and data [83,97].

The terms "adapter" and "wrapper" are often used interchangeably, as we will also do throughout this article. Another term that is often adopted in the same context of that of wrappers, is the term "connector". A connector refers to the encapsulation of a communication mechanism. This term originates from architectural description languages (ADLs), while the design pattern which is concerned with the encapsulation of the communication between components is called "Mediator" [20]. The prime distinction between a connector and an adapter is that an adapter bridges an interoperability problem, i.e., without the adapter the components would not be able to work together, while a connector enables the communication between two interoperable components [95].

Simply speaking, a component wrapper contains a software layer that encapsulates legacy data and logic and defines its services in the wrapper API. The wrapper API mediates between calls from client application components to the underlying legacy code and data, by transforming incoming requests into a message format that is understandable to the internal code and data.

Execution in this type of architecture occurs among component wrappers within the application server. These component wrappers wrap back-end resources such as databases, ERP, transaction mainframe systems and legacy systems so that they can express data and messages in the standard internal format expected by the application server. The application server is oblivious to the fact that these components are only the external facade of existing EIS that do the real processing activities [63].

The adapter/component wrapper pairs in the architecture illustrated in Fig. 7 are responsible for providing a layer of abstraction between the application server and the component EIS. This layer allows for EIS component communications, as if the component EIS were executed within the application server environment itself. Traditional application servers constitute an ideal

support framework for synchronous business processes. However, newer generation application servers offer also asynchronous communication.

Web application servers already provide database connectivity, transaction management, EAI-style connectors, message queuing, and are gradually evolving into business process execution engines. They also facilitate reliability, scalability, and availability, while at the same time automating application development tasks.

In concluding, a few words about application server implementation. Application servers are principally J2EE-based and include support for JMS [45], the Java 2 Connector Architecture (J2CA) [50], and even Web services.

JMS is a transport-level API that enterprises can combine with Web service solutions for messaging, data persistence, and access to Java-based applications. JMS is a vendor agnostic API for enterprise messaging that can be used with many different MOM vendors. JMS frameworks function in asynchronous mode but also offer the capability of simulating a synchronous request/response mode [70]. For application server implementations, JMS provides access to business logic distributed among heterogeneous systems. Having a message-based interface enables point to point and publish/subscribe mechanisms, guaranteed information delivery, and interoperability between heterogeneous platforms.

J2EE Connector Architecture is an emerging technology that has been specifically designed to address the hardships of integrating applications. J2CA provides a standardized method for integrating disparate applications in J2EE application architectures. It provides support for resource adaptation, which maps the J2EE security, transaction, and communication pooling to the corresponding EIS technology. J2CA defines a set of functionality that application server vendors can use to connect to back-end EIS, such as ERP, CRM, and legacy systems and applications. Using J2CA to access EIS is akin to using JDBC (Java Database Connectivity) [101] to access a database. When J2CA is used in an ESB implementation, the ESB could provide a J2CA container that allows packaged or legacy applications to be plugged into the ESB through J2CA resource adapters. For instance, a process order service uses JCA to talk to a J2EE application that internally fulfils incoming orders. The latest versions of many application servers, including BEA WebLogic and IBM WebSphere, support J2CA adapters for enterprise connectivity. In addition, major packaged application vendors have also announced plans to support JCA in future product offerings. The ESB uses J2CA to facilitate application integration between existing applications and services.

3.4.3 Business process management

Today enterprises are striving to become electronically connected to their customers, suppliers, and partners. To achieve this, they are integrating a wide range of discrete business processes across application boundaries of all kinds. Application boundaries may range from simple enquiries about a customer's order involving two applications, to complex, long-lived transactions for processing an insurance claim involving many applications and human interactions, to parallel business events for advanced planning, production and shipping of goods along the supply chain involving many applications, human interactions and business to business interactions. When integrating on such a scale, enterprises need a greater latitude of functionality to overcome multiple challenges arising from the existence of proprietary interfaces, diverse standards, and approaches targeting the technical, data, automated business process, process analysis, and visualization levels. Such challenges are addressed by the business process management (BPM) technology [96]. BPM is the term used to describe the new generation of technology that provides end-to-end visibility and control over all parts of a long-lived, multi-step information request or transaction/process that spans multiple applications and human actors in one or more enterprises [60].

Specialized capabilities of BPM software solutions in an ESB setting include workflow-related business processes, process analysis, and visualization techniques. In particular, BPM allows the separation of business processes from the underlying integration code. Before we explain further characteristics and typical elements of BPM, it is useful to distinguish between the concepts of process automation, workflow, and business processes management.

All enterprises have business processes that require process automation. Any process automation tool should be able to easily control and coordinate activity and provide an easy method to define the business process and the underlying flows of information between applications. Process automation is distinct from traditional document workflow as it involves integration between computer-based systems and manual steps and tasks. It is implemented for automating the flow of information between applications to fulfil business processes. Traditional workflow tools focus instead on handling the movement of documents between people who are required to perform tasks on these documents [59]. This may or may not be directly associated with managing a business process. For example, an order may generate a shipping notice. The shipping notice may in turn

generate a monthly invoice that encompasses many orders for the same customer. This is an example of a workflow that changes focus several times. It is not tracking the order to completion, but rather, it tracks only the information generated.

A business process includes both automated and manual processes. Business process automation combines process automation together with task-based workflow into a managed, end-to-end process [59].

Business Process Management codifies value-driven processes and institutionalizes their execution within the enterprise [80]. This implies that BPM tools, such as Chordiant,[1] Pega [2] and Fuego,[3] can help analyze, define, and enforce process standardization. BPM provides a modeling tool to visually construct, analyze, and execute cross-functional business processes. Design and modeling of business processes is accomplished by means of sophisticated graphical tools. In the previous example, BPM would enable the modeling of the broader order management process encompassing order receipt, perhaps credit approval, shipping and invoicing. Combining BPM with real-time analysis allows business managers to not only track where orders are in this process, but also to understand the company's exposure with regard to orders in total at any given point in time.

Business Process Management is a commitment to expressing, understanding, representing and managing a business (or the portion of business to which it is applied) in terms of a collection of business processes that are responsive to a business environment of internal or external events [67]. The term management of business processes includes process analysis, process definition and redefinition, resource allocation, scheduling, measurement of process quality and efficiency, and process optimization. Process optimization includes collection and analysis of both real-time measures (monitoring) and strategic measures (performance management), and their correlation as the basis for process improvement and innovation.

Business Process Management is driven primarily by the common desire to integrate supply chains, as well as internal enterprise functions, without the need for even more custom software development. This means that the tools must be suitable for business analysts, requiring less (or no) software development. They must reduce maintenance requirements because internally and externally integrated environments routinely require additions and changes to business processes. BPM promises the ability to monitor both the state of any single process instance and all instances in the aggregate, using present real-time metrics that translate actual process activity into key performance indicators.

When sophisticated process definitions are called for in an ESB, a process orchestration engine—that supports BPEL [6] or some other process definition language such as ebXML Business Process Specification Schema (BPSS) [21]—may be layered onto the ESB. The process orchestration may support long-running, stateful processes, just like BPEL. In addition, it may support parallel execution paths, with branching, and merging of message flow execution paths based on join conditions or transition conditions being met. Sophisticated process orchestration can be combined with stateless, itinerary-based routing to create an SOA that solves complex integration problems. An ESB uses the concept of itinerary-based routing to provide a message with a list of routing instructions. In an ESB, routing instructions, which represent a business process definition, are carried with the message as it travels through the bus across service invocations. The remote ESB service containers determine where to send the message next, making this type of routing a special category of content-based routing.

The ESB can also benefit from products developed by BPM vendors such as IBM's WebSphere, HP's HP Process Manager, BEA's WebLogic, and Vitria's BusinessWare. Microsoft BizTalk is another good example of a BPM integration product, but its use is limited to Microsoft Windows and .NET servers. Commercial BPM solutions such as, for instance, Vitria's BusinessWare-4 provide organizations with a number of mechanisms for simplifying the deployment and management of an integration solution. They consist of range of tools and technologies that allow the organization to model, test, deploy, and refine such a process-driven integration solution.

3.5 Adapters

For the most part, business applications in an enterprise are not designed to communicate with other applications. There is often an interoperability mismatch between the technologies used within internal systems and with external trading partner systems. In order to seamlessly integrate these disparate applications, there must be a way in which a request for information in one format can easily be transformed into a format expected by the called service. For instance, in Fig. 3 the functionality of a J2EE application needs to be exposed to non-J2EE clients such as .NET applications and other clients. In doing so, a Web service may have to integrate with other instances of EIS in an organization,

[1] www.chordiant.com.

[2] www.pega.com.

[3] www.fuego.com.

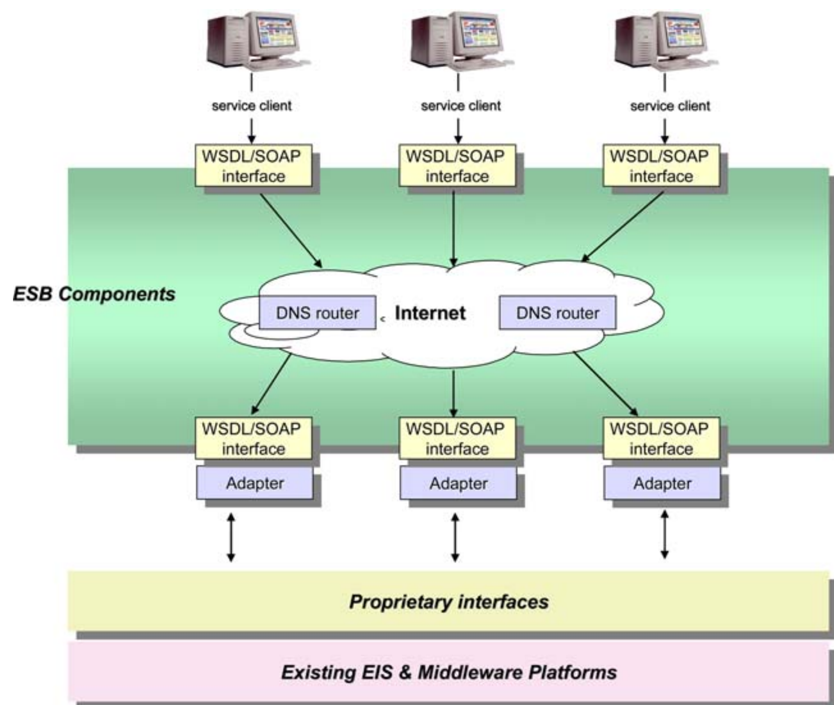

Fig. 8 Combining Web services with resource adapters

or the J2EE application itself may have to integrate with other EISs. In such scenarios, how the application exchanges information to the ESB depends on the application accessibility options. There are three alternative ways an application can exchange information with the ESB include [52]:

1. *Application-provided Web service interface* Some applications and legacy application servers have adopted the open standards philosophy and have included a Web services interface. The WSDL defines the interface to communicate directly with the application business logic. Where possible, taking a direct approach is always preferred.
2. *Non-Web service interface* The application does not expose business logic via Web services. An application-specific adapter can be supplied to provide a basic intermediary between the application API and the ESB.
3. *Service wrapper as interface to adapter* In some cases the adapter may not supply the correct protocol (JMS, for example) that the ESB expects. In this case, the adapter would be Web service enabled.

Adapters provide connectivity, semantic disambiguation and translation services between applications and collaborations [84]. An adapter provides services between the integration broker and the application-specific component, such as an ERP or CRM application. Resource adaptors translate the applications' messages to and from a common set of standards—standard data formats and standard communication protocols. When an application sends a message to another application, its adapter first translates the message into a standardized form. When the message is received by the target application, another adapter translates the standardized message into the target application's native format and protocol.

An adapter can expose either a synchronous and asynchronous mode of communication between the client application and the EIS. The adapter provides a variety of transformation services including support for complex data structures from one data source to another, for example, between COBOL copybooks and XML, complex XML-to-XML vocabularies, IDL, ODL, legacy, database systems, and so on. Messages are first transformed into an intermediate state (IDL, XML, or Java Interfaces) before being formatted. An adapter typically provides support for a range of date/time conversion functions, EBCDIC/ASCI, binary and character conversion functions, EDI (via EDI module), and a number of functions for splitting/combining data fields. Furthermore, it can incorporate a native XML parser and provides support for popular e-Commerce trading protocols such as RosettaNet [104], ebXML [38], cXML,[4] and xCBL.[5]

[4] http://www.cxml.org.

[5] http://eco.commerce.net.

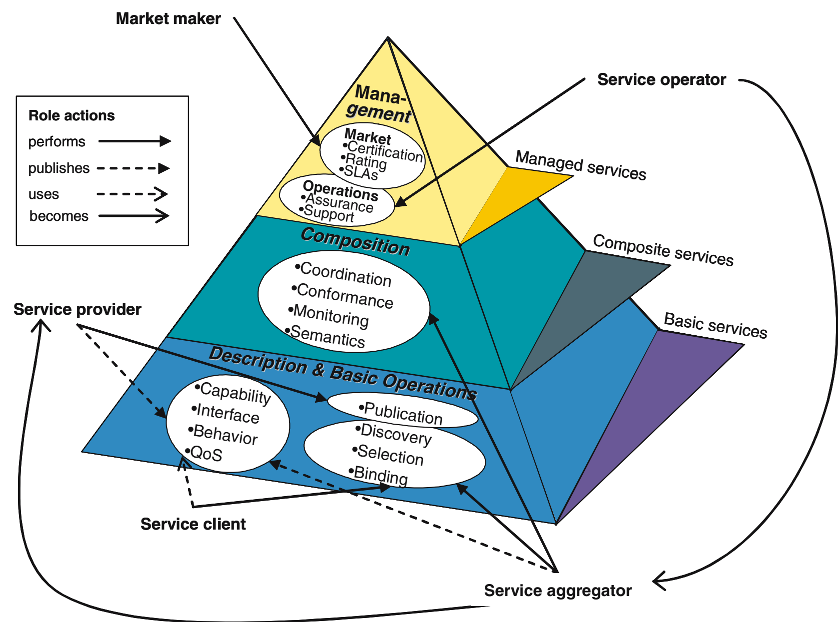

Fig. 9 Extended SOA

As complementary technologies in an ESB implementation, (resource) adapters and Web services can work together to implement complex integration scenarios involving federated ESBs spanning multiple organizations, see Fig. 8. Data synchronization (in addition to translation services) is one of the primary objectives of resource adapters. Adapters can thus take on the role of data synchronization and translation services, whereas Web services will enable application functions to interact with each other. Web services are an ideal mechanism for implementing a universally accessible application function (service) that may need to integrate with other applications to fulfil its service contract. The drivers of data synchronization and Web services are also different. Web services will generally be initiated by a user request/event, whereas data synchronization is generally initiated by state changes in data objects (for example, customer, item, order, and so on).

An event to which a Web service reacts could be a user initiated request such as a purchase order or an online bill payment, for example. User events can naturally be generated by applications such as an order management application requiring a customer status check from an accounting system. On the other hand, a state change in a data object can be an activity like the addition of a new customer record in the customer service application or an update to the customer's billing address. These state changes trigger an adapter to add the new customer record or update the customer record in all other applications that keep their own copies of customer data.

Routing of events from service requester to service providers may basically occur in two ways, using content-based or topic (subject)-based routing (see Sect. 3.4.1). Both routing mechanisms run on top of elementary Internet-technologies for routing, e.g., DNS routing. Currently, routing of events is standardized in WS-Notification.

Reverting to the J2EE to .NET application connectivity scenario, a connectivity service in the form of a resource adapter is required. In this implementation strategy, Web services can become the interface between the company and its customers, partners, and suppliers; whereas the resource adapters become integration components tying up different EISs inside the company. This is just one potential implementation pattern in which Web services and resource adapters can coexist. Another potential integration pattern in which Web services and resource adapters are required to collaborate is in business process integration. Applications that are part of a specific business process will have to expose the required processes (functions), and Web services are ideal for that purpose. When the applications need to integrate with other EISs to fulfil their part in the business process, they will use resource adapters.

4 Extending the SOA

A basic SOA, i.e., the architecture depicted in Fig. 2, implements concepts such as service registration, discovery, load balancing of service requests. The essential ESB requirements, however, suggest that this approach be extended to support capabilities such as service

orchestration, "intelligent" routing, provisioning, and service management. It should also guarantee the integrity of data and security of messages. Such overarching concerns are addressed by the extended SOA (xSOA) [73,75]. The xSOA is an attempt to streamline, group together and logically structure the functional requirements of complex applications that make use of the service-oriented computing paradigm. The xSOA is a stratified service-based architecture. The architectural layers in the xSOA, which are depicted in Fig. 9, embrace a multi-dimensional, separation of concerns [72] in such a way that each layer defines a set of constructs, roles, and responsibilities and leans on constructs of its predecessor layer to accomplish its mission. The logical separation of functionality is based on the need to separate basic service capabilities provided by the conventional SOA (for example, building simple applications) from more advanced service functionality needed for composing services and the need to distinguish between the functionality for composing services from that of the management of services. As shown in Fig. 9, the xSOA utilizes the basic SOA constructs as its foundational layer and layers service composition and management on top of it. The ESB middleware capabilities such as communication, routing, translation, discovery, and so on, fall squarely within the xSOA foundation layer. ESB capabilities that deal with service composition and management can be found in the composition and management layers of the xSOA. However, these layers include more advanced functionality than that found in ESB settings.

The bottom layer in the xSOA is identical to the basic SOA (see Fig. 2) in that it defines an interaction between software agents as an exchange of messages between service requesters (clients) and service providers. These interactions involve the publishing, finding and binding of operations.

In a typical service-based scenario employing the basic services layer in the xSOA, a service provider hosts a network accessible software module. The service provider defines a service description, and publishes it to a client (or service discovery agency) through which a service description is advertised and made discoverable. The service client (requestor) discovers a service (endpoint) and retrieves the service description directly from the service (Metadata Exchange) or from a registry or repository like UDDI; it uses the service description to bind with the service provider and invoke the service or interact with the service implementation. Service provider and service client roles are logical constructs and a service may exhibit characteristics of both. For reasons of conceptual simplicity in Fig. 9, we assume that service clients, providers and aggregators could act as service brokers or service discovery agencies and publish the services they deploy. The role actions in this figure also indicate that a service aggregator entails a special type of provider.

The service composition layer in the xSOA encompasses necessary roles and functionality for the aggregation of multiple services into a single composite service. Resulting composite services may be used by service aggregators as basic services in further service compositions or may be utilized as applications/solutions by service clients. Service aggregators thus become service providers by publishing the service descriptions of the composite service they create. As already explained in Sect. 2, a service aggregator is a service provider that groups services that are provided by other service providers into a distinct value added service. Service aggregators develop specifications and/or code that permit the composite service to perform functions that are based on the following facilities (in addition to Web services interoperation support provided by WSI profiles for service descriptions [10] and security [11]):

- *Meta-data, standard terminology and reference models* Web services need to use meta-data to describe what other endpoints need to know to interact with them. Metadata describing a service typically contain descriptions of the interfaces of a service—the kinds of data entities expected and the names of the operations supported—such items as vendor identifier, narrative description of the service, Internet address for messages, format of request and response messages, and may also contain choreographic descriptions of the order of interactions. Such descriptions may range from simple identifiers implying a mutually understood protocol to a complete description of the vocabularies, expected behaviors, and so on. Such meta-data give high-level semantic details regarding the structure and contents of the messages received and sent by Web services, message operations, concrete network protocols, and endpoint addresses used by Web services; it also describes abstractly the capabilities, requirements, and general characteristics of Web services and how they can interoperate with other services. Web service meta-data need to be accompanied with standard terminology to address business terminology fluctuations and reference models such as, for instance, RosettaNet PIPS [104], to allow applications to define data and processes that are meaningful not only to their own businesses but also to their business partners while also maintaining interoperability (at the semantic level) with other

business applications. The purpose of combining meta-data, standard terminology and reference models is to enable business processes to capture and convey their intended meaning and exchange requirements, identifying among other things the meaning, timing, sequence and purpose of each business collaboration and associated information exchange.

- *Conformance* Service conformance ensures the integrity of a composite service by matching its operations and parameter types with those of its constituent component services, imposes constraints on the component services (e.g., to ensure enforcement of business rules), and performs data fusion activities. Service conformance comprises three parts: typing, behavioral, and semantic conformance. Typing (syntactic) conformance is performed at the data typing level by using principles such as type-safeness, co-variance and contra-variance for signature matching [95]. Behavioral conformance ensures the correctness of logical relationships between component operations that need to be blended together into higher-level operations. Behavioral conformance guarantees that composite operations do not lead to spurious results and that the overall process behaves in a correct and unambiguous manner. Finally, semantic conformance ensures that services and operations are annotated with domain-specific semantic properties (descriptions) so that they preserve their meaning when they are composed and can be formally validated. Service conformance is a topic still under research scrutiny ([95]). Concrete solutions exist only for typing conformance [24,68] as they are based on conformance techniques for programing languages such as Eiffel or Java.
- *Coordination* Controls the execution of the component services (processes), Web services transactions, and manages dataflow as well as control flow among them and to the output of the component service (e.g., by specifying workflow processes and using a workflow engine for run-time control of service execution).
- *Monitoring* Allows monitoring events or information produced by the component services, monitoring instances of business processes, viewing process instance statistics, including the number of instances in each state (running, suspended, aborted, or completed), viewing the status, or summary for selected process instances, suspend, and resume or terminate selected process instances. Of particular significance is the ability to be able to spot problems and exceptions in the business processes and move toward resolving them as soon as they occur. Process monitoring capabilities are currently provided by toolsets in platforms for developing, deploying, and managing service applications, such as, for instance, BEA's WebLogic and Vitria's BusinessWare.
- *Policy enforcement* Web service capabilities and requirements can be expressed in terms of policies. In particular, policies [88] may be applied to manage a system or organize the interaction between Web-services [1]. For example, knowing that a service supports a Web services security standard such as WS-Security is not enough information to enable successful composition. The client needs to know if the service actually requires WS-Security, what kind of security tokens it is capable of processing, and which one it prefers. The client must also determine if the service requires signed messages. And if so, it must determine what token type must be used for the digital signatures. And finally, the client must determine when to encrypt the messages, which algorithm to use, and how to exchange a shared key with the service. Trying to orchestrate with a service without understanding these details may lead to erroneous results.

Standards such BPEL and WS-Choreography [19] that operate at the service composition layer in xSOA enable the creation of large service collaborations that go far beyond allowing two companies to conduct business in an automated fashion. We also expect to see much larger service collaborations spanning entire industry groups and other complex business relationships. These developments necessitate the use of tools and utilities that provide them insights into the health of systems that implement Web services and into the status and behavior patterns of loosely coupled applications. A consistent management methodology is essential for leveraging a management framework for a production-quality, service-based infrastructure, and applications. The rationale is very similar to the situation in traditional distributed computing environments, where systems administrators rely on programs/tools/utilities to make certain that a distributed computing environment operates reliably and efficiently.

Managing loosely coupled applications in an SOA inherently entails even more challenging requirements. Failure or change of a single application component can bring down numerous interdependent enterprise applications. The addition of new applications or components can overload existing components, causing unexpected degradation or failure of seemingly unrelated systems. Application performance depends on the combined performance of cooperating components and their

interactions. To counter such situations, enterprises need to constantly monitor the health of their applications. The performance should be in tune, at all times and under all load conditions.

Managing critical Web service based applications requires in-depth administration and management capabilities that are consistent across an increasingly heterogeneous set of participating distributed component systems, while supporting complex aggregate (cross-component) management use cases, like service-level agreement enforcement and dynamic resource provisioning. Such capabilities are provided by the topmost layer in the xSOA.

We could define Web services management as the functionality required for discovering the existence, availability, performance, health, patterns of usage, extensibility, as well as the control and configuration, life-cycle support and maintenance of a Web service or business process within the context of SOAs. Service management encompasses the control and monitoring of SOA-based applications throughout their life cycle [58]. It spans a range of activities from installation and configuration to collecting metrics and tuning to ensure responsive service execution. The management layer in xSOA requires that a critical characteristic be realized: that services be managed. In fact, the very same well-defined structures and standards that form the basis for Web services also provide opportunities for use in managing and monitoring communications between services, and their underlying resources, across numerous vendors, platforms, technologies, and topologies.

Service management includes many interrelated functions [26]. The most prominent functions of service management are summarized in the following:

1. Service-level agreement (SLA) management. This may include QoS (e.g., sustainable network bandwidth with priority messaging service) [46]; service reporting (e.g., acceptable system response time); and service metering.
2. Auditing, monitoring, and troubleshooting. This may include providing service performance and utilization statistics, measurement of transaction arrival rates and response times, measurement of transaction loads (number of bytes per incoming and outgoing transaction), load balancing across servers, measuring the health of services and troubleshooting.
3. Dynamic services (or resources) provisioning. This may include provisioning services and resources to authorized personnel, dynamic allocation/deallocation of hardware, installation/deinstallation of software "on demand" based changing workloads, ensuring SLAs, management policies for messaging routing and security, and reliable SOAP messaging delivery.
4. Service lifecycle/state management. This may include exposing the current state of a service and permit lifecycle management including the ability to start and stop a service, the ability to make specific configuration changes to a deployed Web service, support the description of versions of Web services and notification of a change or impending change to the service interface or implementation.
5. Scalability/extensibility. The Web services support environment should be extensible and must permit discovery of supported management functionality in a given instantiation.

To manage critical applications/solutions and collaborations spanning entire industry groups and other complex business relationships, e.g., specific service markets, the xSOA management services are divided in two complementary categories ([73,75]):

1. Service operations management that can be used to manage the service platform, the deployment of services and the applications and, in particular, monitor the correctness and overall functionality of aggregated/orchestrated services.
2. Service market management that supports typical integrated supply chain functions and provides a comprehensive range of services supporting industry-trade, including services that provide business transaction negotiation and facilitation, financial settlement, service certification and quality assurance, rating services, service metrics, and so on.

The xSOA's service operations management functionality is aimed at supporting critical applications that require enterprises to manage the service platform, the deployment of services and the applications. xSOA's service operations management typically gathers information about the managed service platform, Web services and business processes and managed resource status and performance, and supporting specific management tasks (e.g., root cause failure analysis, SLA monitoring and reporting, service deployment, and life cycle management and capacity planning). Operations management functionality may provide detailed application performance statistics that support assessment of the application effectiveness, permit complete visibility into individual business processes and transactions, guarantee consistency of service compositions, and deliver application status notifications when a particular activity is completed or when a decision condition is reached. We

refer to the role responsible for performing such operation management functions as the service operator. Depending on the application requirements a service operator could be a service client or service aggregator.

Service operations management is a critical function that can be used to monitor the correctness and overall functionality of aggregated/orchestrated services thus avoiding a severe risk of service errors. Considerations need also be made for modeling the context in which a given service is being leveraged individual, composite, part of a long-running business process, and so on. In order to successfully compose Web services processes, one must fully understand the service's WSDL contract along with any additional requirements, capabilities, and policies (preferences). In this way one can avoid typical errors that may occur when individual service-level agreements (SLAs) are not properly matched. Proper management and monitoring provide a strong mitigation of this type of risk, since the operations management level allows business managers to check the correctness, consistency, and adequacy of the mappings between the input and output service operations and aggregate services in a service composition.

It is increasingly important for service operators to define and support active capabilities versus traditional passive capabilities [42]. For example, rather than merely raising an alert when a given Web service is unable to meet the performance requirements of a given service-level agreement, the management framework should be able to take corrective action. This action could take the form of rerouting requests to a backup service that is less heavily loaded, or provisioning a new application server with an instance of the software providing the service if no backup is currently running and available.

Finally, service operations management should also provide global visibility of running processes, comparable to that provided by BPM tools. Management visibility is expressed in the form of real-time and historical reports, and in triggered actions. For example, deviations from key performance indicator target values, such as the percent of requests fulfilled within the limits specified by a service level agreement, might trigger an alert and an escalation procedure.

Another aim of xSOA's service management layer is to provide support for service markets (aka of open service marketplaces). Currently, there exist several vertical industry marketplaces, such as those for the semiconductor, automotive, travel, and financial services industries. Service cooperatives operate much in the same way like vertical marketplaces, however, they are open. Their purpose is to create opportunities for buyers and sellers to meet and conduct business electronically, or aggregate service supply/demand by offering added value services and grouping buying power (just like a co-operative). The scope of such a service market would be limited only by the ability of enterprises to make their offerings visible to other enterprises and establish industry specific protocols by which to conduct business. Service markets typically support supply chain management by providing to their members a unified view of products and services, standard business terminology, and detailed business process descriptions. In addition, service markets must offer a comprehensive range of services supporting industry-trade, including services that provide business transaction negotiation and facilitation [17], financial settlement, service certification and quality assurance, rating services, service metrics such as number of current service requesters, average turn around time, and manage the negotiation and enforcement of SLAs. The xSOA market management functionality as illustrated in Fig. 9 is aimed to support these open service market functions.

Service markets introduce a new role that of a market maker. A market maker is a third trusted party or consortium of organizations that brings the suppliers and vendors together. Essentially, a service market maker is a special kind of service aggregator that has added responsibilities, such as issuing certificates, maintaining industry standard information and introducing new standards, endorsing service providers/aggregators, etc. The market maker assumes the responsibility of the service market administration and performs maintenance tasks to ensure the administration is open and fair for business and, in general, provides facilities for the design and delivery of integrated service offerings that meet specific business needs and conform to industry standards.

5 Research activities and open issues

This section focuses on ongoing research activities conducted on services. Also, it identifies open research issues. We classify both the research activities and open research issues on the basis of the functional layers of xSOA.

5.1 xSOA basic services layers

Research activities in the basics services layer to date target formal service description language(s) for holistic service definitions addressing, besides functional aspects, also behavioral as well as non-functional aspects associated with services. They also concentrate on open, modular, extensible framework for service discovery, publication and notification mechanisms across distrib-

uted, heterogeneous, dynamic (virtual) organizations as well as unified discovery interfaces and query languages for multiple pathways. In the following, we summarize several research activities contribute to these and related problems.

In addition to the application-specific functions that services provide, services may also support (different) sets of protocols and formats addressing extra-functional concerns such as transaction processing and reliable messaging.

Tai et al. [92] address the problem of transactional coordination in service-oriented computing. The authors of this publication argue for the use of declarative policy assertions to advertise and match support for different transaction styles (direct transaction processing, queued transaction processing, and compensation-based transaction processing) and introduce the concept of and system support for transaction coupling modes as the policy-based contracts guiding transactional business process execution.

An SOA requires that developers discover at development time service descriptions in (UDDI) repository systems and, by reading these descriptions they are able to code client applications that can (at run time) bind to and interact with services of a specific type (i.e., compliant to a certain interface and protocol). Understanding the execution semantics is a rather cumbersome task.

To address this problem Deora et al. ([34,35]) propose a quality of service management framework based on user expectations. This framework collects expectations as well as ratings from the users of a service and then the quality of the service is calculated only at the time a request for the service is made and only by using the ratings that have similar expectations. Similar research efforts are reported in [78].

The AI and semantic Web community has concentrated their efforts in giving richer semantic descriptions of Web services that describe the properties and capabilities of Web services in an computer-interpretable form. For this purpose, DAML-S [93] language has been proposed to facilitate the automation of Web service tasks including better means of Web service discovery, execution, "automatic" composition, verification, and execution monitoring.

In addition, in [77], an approach is described that builds on top of existing Web services technologies and combines them with some concepts borrowed from the Semantic Web to leverage Web service discovery and composition. This approach is captured by the METEOR-S Web Service Annotation Framework (MWSAF). In particular, the MWSAF is designed to semi-automatically mark up Web service descriptions with ontologies.

In the basic SOA UDDI provides a simple browsing-by-business-category mechanism for developers to review and select published services. It is generally believed that discovery based on keyword-search could be improved considerably by introducing more powerful matching approaches. In [95], a hybrid matching approach is suggested, combining semantic and syntactic comparison algorithms of WSDL documents. Comparable research efforts have been reported in [37,51,100].

In order for SOA to become successful, powerful mechanisms are needed that allow service requestors to find service providers that are able to provide the services they need. Typically, this service trading needs to be executed in several stages as the offer descriptions are not completely specified in most cases and different parameters have to be supplemented by the service requestor and provider alternately. Unfortunately, existing service description languages (like DAML-S) treat service discovery as a one-shot activity rather than as an ongoing process and accordingly do not support this stepwise refinement.

Klein et al. [53] introduce the concept of partially instantiated service descriptions containing different types of variables which are instantiated successively, thereby mirroring step-by-step progress in a trading process. The suggested approach is grounded on service ontologies that were developed in the DIANE project [54].

In [81], a peer-to-peer based framework is investigated that allows to advertise and find services using keyword-based search, ontology-based search and behavior-based search in a highly decentralized and dynamic environment. In addition, the framework provides mechanisms so that users may express and query the quality of services.

5.2 xSOA composition layer: research activities

Service composition is today largely a static affair. All service interactions are anticipated in advance and there is a perfect match between output and input signatures and functionality. More ad hoc and dynamic service compositions are required very much in the spirit of lightweight and adaptive workflow methodologies. These methodologies will include advanced forms of co-ordination, less structured process models, and automated planning techniques as part of the integration/composition process. On the transactional front, although standards like WS-Transaction, WS-Coordination, and BTP are a step in the right direction, they fall short of describing different types of atomicity needs for e-business and e-government applications. These do

not distinguish between transaction phases and conversational sequences, e.g., negotiation. Another area that is lacking research results is advanced methodologies in support for the service composition lifecycle. Several research activities contribute to these and related problems.

Yang and Papazoglou [102] present an integrated framework and prototype system that manage the entire life-cycle of service components ranging from abstract service component definition, scheduling, and construction to execution. Service compositions are divided in three categories: fixed, semi-fixed, and explorative compositions. Fixed service compositions require that their constituent services be synthesized in a fixed (pre-specified) manner. Semi-fixed compositions require that the entire service composition is specified statically but the actual service bindings are decided at run time. Finally, explorative compositions are generated on the fly on the basis of a request expressed by a client (application developer).

In [90], a framework is introduced for enriching semantic service descriptions with two compositional assertions: assumption and commitment that help to reason about service composition and verification. The framework uses the interval temporal logic (ITL) language for representing and proving temporal properties of systems. This framework is embedded in the Semantic Web rule language [48], which combines Horn-like rules with an OWL knowledge base.

Charfi and Mezini [29] propose to modularize Web-service composition adopting two dimensions, one for outlining the business process flow, using languages such as BPEL and BPL, and a second dimension comprising business rules, which may be attached to the business processes and evolve independently.

There has been some work in the area of applying AI planning techniques to automate the composition of Web Services. In [86], the authors describe how an OWL reasoner can be integrated with an AI planner to overcome the problem of closed world semantics of planners versus open world semantics of OWL.

Many of the existing approaches toward service composition largely neglect the context in which composition takes place. In [32], a context-aware methodology is introduced that considers a social specification of a service composition as the basis for process specification and verification.

5.3 xSOA management layer

Service management constitutes the foundation of the upper layer of the extended SOA. Traditional management applications fail to meet enterprise requirements in a service-centric world. Conventional systems management approaches and products view the world in a very coarse (mostly applications oriented) manner. The most recent wave of management product categories does not have the business awareness that services management will require. The finer grained nature of services (as opposed to applications) requires evaluating processes and transactions at a more magnified rate and in a more contextually aware manner.

One crucial aspect of management entails monitoring. In [12], a dynamic monitoring approach is probed that is capable of specifying monitoring rules governing the control of WS-BPEL processes.

Casati et al. [25] concentrate on operations management. The proposed business oriented management of Web services is an attempt to assess the impact of service execution from a business perspective and, conversely, to adjust and optimize service executions based on stated business objectives. This is a crucial issue as corporations strive to align service functionality with business goals.

In [87] a model-driven trust negotiation framework for Web services management is explored. The framework adopts a model for trust negotiation that employs state machines, which incorporate security policies. The policy model underlying this framework facilitates lifecycle management.

The ability to gauge the quality of a service is critical if we are to achieve the service oriented computing paradigm. Many techniques have been proposed and most of them attempt to calculate the quality of a service by collecting quality ratings from the users of the service, and then combining them in one way or another. Collecting quality ratings alone from the users is not sufficient for deriving a reliable or accurate quality measure for a service.

In [69], Maximilien extends the usage of the Quality of Service concept for not only selecting Web-services but also establishing trust between trading partners. This paper outlines an agent-oriented approach, including an architecture and programing model. The work is validated empirically, based on a series of simulations.

Ideally, services are collaborating in highly distributed environments, naturally cutting across various enterprise boundaries. This environment demands that contracts are set up, stipulating agreements between services regarding their collaboration, both at the functional and non-functional level, in a concise manner. These contracts may serve as the basis for process monitoring and adaptation.

Ludwig et al. (IBM) [65] suggest to standardize on agreements between enterprise domains, proposing the

WS-Agreement standard. Their work addresses both the contract creation and monitoring from the perspective of service providers and consumers. Service providers are supported by an infrastructure offering agreement templates, and facilities to dynamically check the state of an agreement. Likewise, service requesters are in need of contract templates and some monitoring capacities. Lastly, the paper presents the creation and monitoring of agreements (CREMONIA) architecture, implementing WS-Agreements.

6 Summary

Modern enterprises need to align into virtual alliances, while responding effectively and swiftly to competitive challenges. Therefore, they are required to streamline both internal and external business processes by integrating the various packaged and home-grown applications found spread throughout an enterprise. This requires an agile approach that allows enterprise business services (those offered to customers and partners) to be easily assembled from a collection of smaller, more fundamental business services. This challenge in automated business integration has driven major advances in technology within the integration software space. As a result the SOA has emerged recently, essentially addressing the requirements of service requesters, providers and service brokers, regarding loosely coupled, standards-based, and protocol-independent distributed computing and offering ways to achieve the desired levels of business integration effectively, mapping IT implementations more closely to the overall business process flow.

Combining the SOA paradigm with event-driven processing lays the foundation for an emerging technology, that amalgamates various conventional distributed computing, middleware, BPM and EAI technologies, and thereby offers a unified backbone on top of which enterprise services can be advertised, composed, planned, executed, monitored, and decommissioned. This overarching implementation backbone to SOA is referred to as the ESB.

To cater for essential ESB requirements that include capabilities such as service orchestration, "intelligent" routing, provisioning, integrity and security of message as well as service management, the basic service description/publication/discovery functions of the conventional SOA need to be extended into the extended SOA (xSOA). Particularly, the xSOA incorporates a service composition tier to offer necessary roles and functionality for the consolidation of multiple services into a single composite service. In addition, xSOA provides a separate tier for service management that can be used to monitor the correctness and overall functionality of aggregated/orchestrated services, supporting complex aggregate (cross-component) management use cases, such as service-level agreement enforcement and dynamic resource provisioning.

This paper has surveyed approaches, technologies, and research issues related to services architectures and underlying technologies. Particularly, it has reviewed several technologies, including integration brokers, business process management, and application servers that implement the backbone of an Enterprise Service Bus, which is of critical importance to make the service-oriented computing paradigm operational in a business context.

References

1. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services: Concepts, Architectures and Applications. Springer, Heidelberg (2004)
2. Anagol-Subbaro A.: J2EE Web Services on BEA WebLogic. Prentice-Hall, Upper Saddle River (2005)
3. Anderson, S., et al.: Web Services Trust language (WS-Trust). Public draft release, Actional Corporation, BEA Systems, Computer Associates International, International Business Machines Corporation, Layer 7 Technologies, Microsoft Corporation, Oblix Inc., OpenNetwork technologies, Ping Identity, Reacticity, RSA Security, and Verisign, February (2005)
4. Andrews, T., et al.: Business process execution language (BPEL), version 1.1. Technical-report, BEA Systems and International Business Machines Corporation, Microsoft Corporation, SAP AG and Siebel Systems, May 2003
5. Arkin, A.: Business process modeling language (BPML). Last Call Draft Report, BPMI.Org, November 2002
6. Arora, A., et al.: Web services for management (WS-Management). Technical report, Advanced Micro Devices, Dell, Intel, Microsoft Corporation and Sun Microsystems, October 2004
7. Arsanjani, A.: Introduction to the special issue on developing and integrating enterprise components and services. Commun. ACM **45**(10), 30–34 (2002)
8. Atkinson, B., et al.: Web Services Security (WS-Security). Technical report, Microsoft, IBM and Verisign, April 2002
9. Bajaj, S., et al.: Web Services Policy framework (WS-Policy). Technical report, BEA Systems Inc., International Business Machines Corporation, Microsoft Corporation, Inc., SAP AG, Sonic Software, and VeriSign Inc., September 2004
10. Ballinger, K., et al.: Web services-interoperability (WSI), Basic profile version 1.1, 2004-08-24. Technical report, WSI Organization (WS-I), 2004
11. Barbir, A., et al.: Basic security profile, version 1.0. Technical report, Web Services-Interoperability Organization (WS-I), 2004
12. Baresi, L., Guinea, S.: Towards dynamic monitoring of WS-BPEL processes. In: Proceedings of the Third International Conference on Service Oriented Computing, pp. 269–282. Springer, Amsterdam (2005)
13. Bloomberg, J.: Events vs. services. Available at: http://www.zapthink.com, ZapThink white paper, October 2004
14. Boag, S., et al.: Xquery 1.0: An XML query language, W3C working draft. Technical report, W3C, April 2005

15. Booth, D., et al.: Web Service Architecture. http://www.w3.org/tr/ws-arch/, W3C, Working Notes, 2003/2004
16. Box, D., et al.: Simple Object Access Protocol (SOAP), Version 1.1. W3C Note, W3C, May 2000. http://www.w3.org/TR/SOAP/
17. Bui, T., Gachet, A.: Web services for negotiation and bargaining in electronic markets: Design requirements and implementation framework. In: Proceedings of the 38th Hawaii International Conference on System Sciences, IEEE, 2005
18. Burbeck, S.: The tao of e-business services: the evolution of Web applications into service-oriented components with Web services. IBM DeveloperWorks, 2000. http://www-106.ibm.com/developerworks/Webservices/library/ws-tao/
19. Burdett, D., Kavantzas, N.: WS-Choreography Model Overview. W3c working draft, W3C, March 2004
20. Buschmann, F., et al.: Pattern-oriented software architecture: a system of patterns. Wiley, New York (1996)
21. Business Process Project Team: ebXML Business Process Specification Schema, version 1.01. OASIS, 2001. http://www.ebxml.org/specs/ebbpss.pdf
22. Bussler, C.: B2B Integration: Concepts and Architecture. Springer, Berlin (2003)
23. Candadai, A.: A dynamic implementation framework for SOA-based applications. Web Logic Dev. J. WLDJ **September/October**, 6–8 (2004)
24. Cardelli, L., Wegner, P.: On understanding types, data abstraction and polymorphism. ACM Comput. Surv. **17**(4), 211–221 (1985)
25. Casati, F. *et al.*: Business-oriented management of Web services. Commun. ACM **46**(10), 55–60 (2003)
26. Catania, N., et al.: Web services management framework, version 2.0. Technical report, HP, July 2003
27. Chappell, D.: Enterprise Service Bus. O'Reilly Media, Inc., Sebastopol (2004)
28. Chappell, D.: ESB myth busters: Clarity of definition for a growing phenomenon. Web Serv. J. February, pp. 22–26 (2005)
29. Charfi, A., Mezini, M.: Hybrid Web service composition: business processes meet business rules. In: ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing, pp. 30–38. ACM Press, New York (2004)
30. Christensen, E., et al.: Web Services Description Language (WSDL) 1.1. W3C Note, W3C, March 2001. http://www.w3.org/TR/2001/NOTE-wsdl-20010315
31. Colan, M.: Service-Oriented Architecture expands the vision of Web services, Part 2. IBM DeveloperWorks, April 2004
32. Colobo, E., Mylopoulos, J., Spoletini, P.: Modeling and Analyzing Context-Aware Composition of Services. In: Proceedings of the Third International Conference on Service Oriented Computing, pp. 198–213. Springer, Amsterdam (2005)
33. Dan, A. et al.: Web services on demand: WSLA-driven automated management. IBM Syst. J. **43**(1), 136–158 (2004)
34. Deora, V., et al.: A quality of service management framework based on user expectations. In: Proceedings of the First International Conference on Service Oriented Computing (ICSOC03), Springer, Heidelberg (2003)
35. Deora, V., et al.: Incorporating QoS specifications in service discovery. In: Proceedings of WISE Workshops, Lecture Notes of Springer Verlag, 2004
36. Dhesiaseelan, A., Ragunathan, V.: Web Services Container Reference Architecture (WSCRA). In: Proceedings of the International Conference on Web Services, IEEE, pp. 806–805, 2004
37. Ding, X., et al.: Similarity search for Web services. In: Proceedings of the 30th VLDB Conference, pp. 372–383, 2004
38. ebXML Technical Architecture Project Team: ebXML Technical Architecture Specification, v1.0.4. Technical report, ebXML.org, February, 2001
39. Farell, S., et al.: Assertions and protocol for the OASIS security assertion markup language (SAML), V1.1. Committee specification, OASIS, July 2003
40. Francis, T., et al.: Professional IBM WebSphere 5.0 Application Server. Wrox, 2002
41. Fremantle, P., Weerawarana, S., Khalaf, R.: Enterprise services. Commun. ACM **45**(10), (2002)
42. Ganek, A.G., Corbi, T.A.: The dawning of the autonomic computer era. IBM Syst. J. **42**(11), 5–18 (2003)
43. Graham, S., et al.: Web Services Resource (WS-Resource), version 1.2, working draft 03. Technical report, OASIS, March 2005
44. Graham, S., Niblett, P.: Web Services Base Notification, version 1.0. Akamai Technologies, Computer Associates International, Inc., Fujitsu Limited, Hewlett-Packard Development Company, International Business Machines Corporation, SAP AG, Sonic Software Corporation, The University of Chicago and Tibco Software Inc., 2004
45. Hapner, M., et al.: Java Messaging Service, version 1.1. Sun technical report, specification, SUN Microsystems, April 2002
46. Hauck, R., Reiser, H.: Monitoring quality of service across organizational boundaries. In: Trends in Distributed Systems: Torwards a Universal Service Market. Proceedings of the third International IFIP/GI Working Conference, 2000
47. Holley, K., Channabasavaiah, K., Tuggle, E.M., Jr.: Migrating to a Service-Oriented Architecture. IBM DeveloperWorks, December 2003
48. Horrocks, I., et al.: SWRL: A Semantic Web Rule Language combining OWL and RULEML, W3C member submission. W3C, 21 May 2004
49. Iwasa, K. (principal ed.) WS-Reliability, version 1.1, committee draft 1.086, 24 august 2004. http://www.oasis-open.org/committees/wsrm/documents/specs/(tbd), OASIS, Web Services Reliable Messaging TC, 2004
50. J2CA Group.: J2EE Connector Architecture Specification, version 1.5. Technical report, SUN Microsystems, 2003
51. Jaeger, M.C., Tang, S.: Ranked matching for service descriptions using DAML-S. In: Grundspenkis, J., Kirikova, M., (eds.), Proceedings of CAiSE'04 Workshops, pp. 217–228. Riga Technical University Riga, Latvia, June 2004
52. Keen, M., et al.: Patterns: Implementing an SOA using an Enterprise Service Bus. IBM Redbook, 22 July 2004
53. Klein, M., Konig-Ries, B., Obreiter, P.: Stepwise refinable service descriptions: Adapting DAML-S to staged service trading. In: Proceedings of 1st International Conference on Service Oriented Computing, December 2003
54. Klein, M., Konig-Ries, B., Mussig, M.: What is needed for Semantic Service Descriptions? A proposal for suitable language constructs. Int. J. Web Grid Serv. **2**, (2005)
55. Krafzig, D., Banke, K., Slama, D.: Enterprise SOA: Service Oriented Architecture Best Practices. Prentice-Hall, Englewood Cliffs (2005)
56. Kreger, H.: Fulfilling the Web services promise. Commun. ACM **46**(6), 29–ff (2003)
57. Kumar, S., Rana, R.: Service on demand portals: a primer on federated portals. Web Logic Dev. J. WLDJ, **September/October**, 22–24 (2004)
58. Lazovik, A., et al.: Associating assertions with business processes and monitoring their execution. In: Proceedings of the Second International Conference on Service Oriented Computing, 2004
59. Leymann, F., Roller, D.: Production Workflow: Concepts and Techniques. Prentice-Hall, Englewood Cliffs (2000)

60. Leymann, W.F., Roller, D., Schmidt, M.-T.: Web services and business process management. IBM Syst. J. **41**(2), 198–211 (2002)
61. Linthicum, D.: Next Generation Application Integration: From Simple Information to Web Services. Addison-Wesley, Reading (2003)
62. Lowey, J.: Programming .NET Components, Reading (2003) 1st edn. O'Reilly Sebastopol (2003)
63. Lubinsky, B., Farrel, M.: enterpise architecture and J2EE. EAI J., pp. 12–15 **November** (2001)
64. Luckham, D.: The Power of Events. An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley, Reading, April 2002
65. Ludwig, H., Dan, A., Kearney, R.: Crona: an architecture and library for creation and monitoring of WS-Agreements. In: ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing, pp. 65–74. ACM Press, New York (2004)
66. Martin, J., Arsanjani, A., Tarr, P., Hailpern, B.: Web Services: Promises and Compromises. Queue, ACM **1**(1), 48–58 (2003)
67. McGoveran, D.: An introduction to BPM and BPMS. Bus. Integr. J., pp. 2–10 **April** (2004)
68. Meyer, B.: Object-oriented Software Construction, 2nd edn. Prentice-Hall Professional Technical Reference, Englewood Cliffs (1997)
69. Maximilien, E.M., Singh, M.P.: Toward autonomic Web services trust and selection. In: ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing, pp. 212–221. ACM Press, New York (2004)
70. Monson-Haefel, R., Chappell, D.: Java Message Services. O'Reilly, 2001.
71. Mukherjee, B., et al.: An efficient multicast protocol for content-based publish-subscribe systems. In: ICDCS '99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, p. 262. IEEE Computer Society, Washington, DC (1999)
72. Ossher, H., Tarr, P.: Multi-dimensional separation of concerns and the hyperspace approach. In: Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development, 2000
73. Papazoglou, M., Georgakopoulos, D.: Introduction to a special issue on service oriented computing. Commun. ACM, **46**(10), 25–28 (2003)
74. Papazoglou, M.P., Ribbers, P.M.A.: e-Business: Organizational and Technical Foundations. Wiley, New York **April** 2006
75. Papazoglou, M.P.: Extending the Service Oriented Architecture. Bus. Integr. J., pp. 18–21 **February** (2005)
76. Parent, C., Spaccapietra, S.: Issues and Approaches of Database Integration. Commun. ACM **41**(5), 166–178 (1998)
77. Patil, A.A., et al.: Meteor-S: web service annotation framework. In: WWW '04: Proceedings of the 13th international conference on World Wide Web, pp. 553–562. ACM Press, New York (2004)
78. Ran, S.: A Model for Web Services Discovery with QoS. SIGecom Exch. **4**(1), 1–10 (2003)
79. Robinson, R.: Understand Enterprise Service Bus scenarios and solutions in Service-Oriented Architecture. IBM DeveloperWorks, June (2004)
80. Roch, E.: Application Integration: Business and Technology trends. EAI J. **August** (2002)
81. Sahin, O.D., et al.: SPiDeR: P2P-Based Web Service Discovery In: Proceedings of the Third International Conference on Service Oriented Computing, pp. 157–170. Springer, Amsterdam (2005)
82. Schulte, R.: Predicts 2003: Enterprise service buses emerge. Report, Gartner, December 2002
83. Seacord, R., et al.: Legacy modernization strategies. Technical Report CMUSEI-2001-TR-025, Carnegie Mellon University, Pittsburgh (2001)
84. Seacord, R.C., Plakosh, D., Lewis, G.A.: Modernizing Legacy Systems. Carnegie Mellon, SEI. Addison-Wesley, Reading (2003)
85. Sing, I., et al.: Designing Web Services with the J2EE 1.4 Platform. Addison-Wesley, Reading (2004)
86. Sirin, E., Parsia, B.: Planning for Semantic Web Services. In: Martin, D., Lara, R., Yamaguchi, T. (eds.) Proceedings of the ISWC 2004 Workshop on Semantic Web Services: Preparing to Meet the World of Business Applications, 2004
87. Skogsrud, H., Benatallah, B., Casati, F.: Trust-serv: model-driven lifecycle management of trust negotiation policies for Web services. In: WWW '04: Proceedings of the 13th international conference on World Wide Web, pp. 53–62. ACM Press, New York (2004)
88. Sloman, M.: Policy driven management of distributed systems. J. Netw. Syst. Manag. **2**, 333–360 (1994)
89. Smith, D.: Web services enable Service Oriented and Event-driven Architectures. Bus. Integr. J., **May**, 12–13 (2004)
90. Solanki, M., Cau, A., Zedan, H.: Augmenting semantic Web service descriptions with compositional specification. In: WWW '04: Proceedings of the 13th international conference on World Wide Web, pp. 544–552. ACM Press, New York (2004)
91. Stal, M.: Web Services: Beyond Component-based Computing. Commun. ACM **45**(10), 71–76 (2002)
92. Tai, S., Mikalsen, T., Wohlstadter, E., Desai, N., Rouvellou, I.: Transaction policies for service-oriented computing. Data Knowl. Eng. **51**(1), 59–79 (2004)
93. The DAML-S Coalition.: DAML-S: Web Service Description for the semantic Web. In: Horrocks, I., Hendler, J.A., (eds.) The Semantic Web - ISWC 2002, First International Semantic Web Conference. Lecture Notes in Computer Science, 2002
94. Universal Description, Discovery, and Integration (UDDI): Technical report, UDDI.ORG, September 2000. http://www.uddi.org
95. van den Heuvel, W.J.: Integrating Modern Business Applications with Legacy Systems: A Software Component Perspective. MIT Press, Cambridge, **February** (2007)
96. van der Aalst, W.M.P.: Lectures on concurrency and petri nets: a tutorial on models, systems and standards for workflow management., In: Business Process Management Demystified, pp. 1–65. Springer, Berlin (2004)
97. Von Schilling, P., Lawrence, P.: Leveraging existing code with object technology. Enterp. Syst. J. **7**, 38–44 (1994)
98. W3C.: XSL Transformations (XSLT), Version 2.0. Technical report, W3C Working Draft, April 2005
99. Wahli, U., et al.: Websphere version 5.1 application developer Web services handbook. IBM Redbook, New York (2004)
100. Wang, Y., Stroulia, E.: Semantic structure matching for assessing Web-service similarity. In: Proceedings of First International Conference on Service Oriented Computing (ICSOC03), pp. 194–207. Springer, Berlin (2003)
101. White, S., Hapner, M.: JDBC 2.1 API. Technical report, SUN, October 1999
102. Yang, J., Papazoglou, M.P.: Service components for managing the life-cycle of service compositions. Inf. Syst. **28**(1), 97–125 (2004)
103. Yang, J.: Web Service Componentization. Commun. ACM **46**(10), 35–40 (2003)
104. Yendluri, P.: RosettaNet implementation framework (RNIF), Version 2.0, Technical report. RosettaNet, 2000