Toward a Programming Model for Service-Oriented Computing

Francisco Curbera, Donald Ferguson, Martin Nally, and Marcia L. Stockton

IBM Corp.
{curbera, dff, nally, mls}@us.ibm.com

Abstract. The service oriented paradigm is, at its core, a model of distributed software components, built around the idea of multi-protocol interoperability and standardized component contracts. The Web Services Interoperability (WS-I) profiles provide standards for runtime interoperability, and the Web Services Description Language (WSDL) and WS-Policy define service contracts that support interoperability between developer tools. A major goal of Service Oriented Architectures (SOAs) is to enable an abstraction layer that integrates and bridges over platform and implementation technology differences, effectively providing a universal business software component and integration framework. Achieving a complete solution requires a portable component model and well-defined patterns for components types. This paper examines the main requirements for a SOA programming model and identifies its most relevant characteristics. In line with SOA's goals, such model must allow a broad community of users (including non-programmers) to create service-oriented applications by *instantiating*, using, *assembling* and *customizing* different *component types* that match the user's goals, skills, and conceptual framework. Moreover, these component types must be portable and interoperable between multiple different vendors' runtimes.

1 Introduction: Service Oriented Architectures

This paper deals with the problem of defining a service-oriented programming model (component model). At its core, a programming model defines

1. A set of *roles,* and skills for each role.
2. A set of *tasks* and an associated role.
3. A set of *part types* or *component types* that the roles create and use.
4. A set of interfaces that a role uses when implementing a specific component type.

As an example, in Java 2 Enterprise Edition™ (J2EE) [1], "dynamic Web page developer" might be a role. A programmer in this role produces Java Server Pages (JSPs) [2] and Servlets [3], and may use JavaBeans™ [4] that encapsulate access to business logic and back-end systems. Programmers in other roles provide the JavaBeans, isolating the dynamic Web page developers from the details of relational database access or integration with non-J2EE applications through connectors and adaptors.

Roles are not necessarily programmer roles, and we use the terms "implement" and "interface" in a loose sense. Defining a programming model has many benefits, most noticeably a reduction in complexity. No single role needs to understand all of the

B. Benatallah, F. Casati, and P. Traverso (Eds.): ICSOC 2005, LNCS 3826, pp. 33 – 47, 2005.
© Springer-Verlag Berlin Heidelberg 2005

possible ways of implementing a function, or all the interfaces a system exports. There are well defined bounds on the breadth of complexity exposed to each role, and well-defined hand-offs between differently skilled developers (different roles). Finally, a programming model enables vendors to provide role and task based tools. The visual metaphors a tool should surface to a programmer implementing a workflow process are significantly different from the metaphors for WYSIWYG Web page design.

This paper focuses primarily on the part types or components of a programming model for Service Oriented Architectures (SOAs). The goals of the service-oriented architecture approach to building enterprise applications include enabling faster integration of business applications inside and between organizations, fostering reuse of application logic, and supporting flexible transformation of enterprise business processes. Taking their cue from the success of the Web in the realm of human-to-application interactions, some say that eventually SOAs should be able to provide support for a new global, fully networked, and dynamic economy.

A precise characterization of SOA may at this point still be a matter of debate. Some key aspects, however, seem to have been widely accepted by now:

1. SOA is a "distributed component" architecture. SOA components are transparently located inside or outside the enterprise and universally accessible as services through a stack of universally supported, interoperable remote procedure call (RPC) and messaging protocols. Standards for defining interfaces provide interoperability between developer tools. "On the wire" protocol interoperability, as opposed to code portability, is the centerpiece of SOA component interactions because it supports the principle of universal access and platform independence. Today, SOA only provides platform independence from the caller's perspective; the service implementer, however, is linked to a specific platform and development tool.
2. Like other component models before it, SOA components encapsulate functionality and enable reuse. However, well-defined SOA components do so at a level granularity and abstraction much closer to the business functions and requirements that are meaningful at the business modeling level (as opposed to the information technology level).
3. SOA components offer declarative, machine processable contracts that enable third parties to access the services that the components provide. SOA contracts explicitly state functional characteristics as well as non-functional (quality of service - QoS) capabilities and requirements. SOA components may document their operations using the Web Services Description Language WSDL [5], and extend this definition to document valid sequences of operations using Business Process Execution Language for Web Services (BPEL4WS) [6] abstract processes.
4. Based on their explicit contracts, components can be automatically and dynamically found, selected and bound by means of their declarative properties, and integrated using composition mechanisms.

The purpose of this paper is to discuss requirements and characteristics of a SOA programming model. Current standards and specifications imply much about the design of the programming model. Four important aspects of a SOA programming model may be derived from the preceding summary characterization of SOA:

1. Platform independence and virtualization.
2. Centrality of composition mechanisms.
3. Flexibility in the component configuration.
4. Loose coupling between components.

We discuss these aspects below.

Virtualization

The central role of universal interoperability in SOA naturally leads to the notion of *virtualization*. From an interoperability standpoint, all applications are accessed as services regardless of their underlying implementation differences and their location in the network (co-resident, inside an enterprise, over the Internet). Likewise, from a *SOA programming model* perspective, applications are (potentially) *SOA components,* despite being implemented in a variety of different underlying technologies.

A SOA programming model in this sense is fundamentally different from other programming models in that it is “virtual” and maps over and into a variety of platform-specific concrete programming models.

Consider two examples:

1. Programmers can use the XSL Transformations (XSLT) language [7] to implement a service that converts the messages used by a “legacy” application to the XML schema defined by an industry standard. The abstraction is portable (XSLT, service invocation). Concrete infrastructures may choose to “compile” the XSLT in Java, C, stored procedure languages, or use an XSLT interpreter.
2. BPEL4WS provides support for defining a service implementation that choreographs and aggregates other services. BPEL4WS *invoke* activity and *partnerLinks* provide a virtual calling mechanism. Other activity types provide support for implementing the service. The programming model is virtual, and specific infrastructures may interpret or compile BPEL4WS as needed.

Although SOA components are not native to any particular platform (.NET, J2EE), applications developed for any platform are potentially SOA components. If the J2EE, .NET, etc. components implement the SOA component model externals (protocols, contracts), other SOA component implementations and solutions can call them.

The preceding examples reveal three aspects of a SOA virtual component and programming model:

1. An abstract primitive for defining requirements on other services (e.g. BPEL4WS partnerLink).
2. An abstraction for calling an operation on a service.
3. A portable abstraction for defining implementation logic (e.g. BPEL4WS, XSLT).

A SOA component model is introduced in Section 2. A virtual component model also requires an abstraction for access to “data” or “information” from within a component’s implementation. In the XSLT example, some of the transformations may require table look-ups. The component is inextricably linked to a specific data access model for data format without such an abstraction. Section 6 describes how Service Data Objects (SDOs) provide this data virtualization layer.

Component Composition
The development of individual (or atomic [1]) service components may rely on platform-specific programming models and languages, or may use an atomic SOA component type like a transform component. Programmers may choose to implement base components using J2EE, PHP [8], etc. A core concern of SOA as a programming model is the interaction of those components and their integration into new composite components or applications. SOA composition may be achieved using platform-specific models, such as a J2EE session bean that accesses back-end services to provide a new service.

SOA-centric composition models, however, can also build directly on top of a SOA component model without mapping into another programming model. BPEL4WS is probably the best known SOA composition language, but different composition models are possible. Most successful composition models will naturally derive from current practice, incorporating proven integration approaches into a SOA programming model.

There are two main perspectives on composition. *Behavioral composition* describes the implementation of the composite; process-oriented composition (derived from workflow models), and a state machine metaphor (such as UML State Diagrams [9]) are good examples of this type of composition. A s*tructural composition,* on the other hand, defines the assembly of a set of existing components into larger solutions. We discuss composition paradigms in Section 3.

Flexibility and Customization
SOA aims to enable the wide reuse of service components. The composition model allows programmers to find services having the desired interfaces and infrastructure (QoS) policies, and aggregate them into new services and solutions. These new services can themselves be composed. It is unlikely, however, that a service can be always be reused "as is", without configuration, customization or tailoring. When change is needed, the current state of the art is source code modification. However, the ability to deliver wide reuse of components depends heavily on the capability to adapt components to the environment in which they are used. A SOA programming model should enable building services and modules that "programmers" can customize without source code modification. This is especially important when the programmer is in a different organization than the programmers who built the components.

In Section 4 we discuss two possible mechanisms for supporting component customization: *adaptation* though *points of variability* (POV) on the component's behavior, and *mediation* which focuses on processing in-flight messages.

Loose Coupling
Another benefit of a SOA programming model is the ability to substitute one component for another at various times during the software lifecycle. This is enabled by the late binding of declared interfaces to implementations supporting them. There are many business reasons why substituting units of functionality is desirable. Most important of these, perhaps, is to reduce the difficulty of managing change in a large

[1] This use of the term atomic is different from the use in transaction processing. In this context, we use atomic to mean a service that is not a composite or aggregate of other components.

enterprise. Being able to introduce change gradually, and limiting the impact of change by adhering to defined interfaces, confers increased flexibility. It also matches the loose coupling that is often characteristic of large human organizations. This feature of a SOA programming model enables groups with different skills, needs and timetables to work collaboratively in a way that maximizes the efficiency of resources, and allows the business to respond more rapidly to change.

There are several elements to loose coupling:

1. Describing messages (operation parameters) using XML Schema makes services less fragile in the face of message evolution. Messages can evolve, for example, through reordering or by adding elements, without breaking existing service implementations. Operation addition or reordering in WSDL does not break existing callers.
2. Dynamic binding is inherently more flexible than existing approaches based on program linking or class paths.
3. The mediation model allows message (request) routing and processing, expanding on the flexibility of dynamic binding. Using routing mediations allows for addition of new or alternate implementations of services, which can be selected during operation invocation based on business logic or rules.

Paper Overview

The rest of this paper examines these aspects of the service oriented architecture from a programming model point of view. Section 2 introduces the notion of SOA components and component types, and discusses some of these component types. Section 3 presents SOA composition models, focusing on structural and behavioral composition paradigms. Section 4 discusses component customization, and in Section 5 we show how data access virtualization is supported by Service Data Objects. Section 6 provides an architectural perspective to the concepts of this paper. In Section 7 we discuss related work and the Service Component Architecture, a recently released SOA programming model. Finally, Section 8 summarizes the results of this paper.

2 SOA Components

Most literature on Web services, especially standards, focuses on the interoperability protocols and service interfaces, and their use. This paper instead focuses on the programming model for *implementing* services and assembling them into solutions. A *component model* simplifies the process of building and assembling services. Here we outline the design of a SOA component model. First, an important distinction between a SOA component and a service must be made. A service is a visible access point to a component. A component can offer multiple services, while at the same time require, as part of its implementation, access to a number or external services. With this distinction in mind, we distinguish three main elements in a SOA component model: service specifications, the service component implementation, and the service component.

A "s*ervice specification*" defines an access channel to a SOA component. It is defined by the following 3 groups of specifications.

- Interfaces, which are typically WSDL portTypes.
- Policies that document QoS properties like transactional behavior, security, etc.
- Behavioral descriptions, for example a BPEL4WS abstract process, or a UML2 state model. Callers can compute valid sequences of operations from the abstract process or state model.

A service specification is different form a Web service in that it is not bound to a network address. An address is assigned to the services provided by a specific component instance, not to the service specifications.

A *service component implementation* is the definition of a particular kind of component, which will in turn admit multiple realizations or instantiations as actual service components. It is defined by 5 groups of specifications.

- Provided "service specifications" define the characteristics of the services that components exposes to potential users.
- Required "service specifications" define the services that the component requires from other service providers to function.
- Properties that may be set on the component to tailor or customize the behavior of each instance of the implementation.
- "Container directives" (policies) that are invariant for all instances of the implementation, including information of the kind typically encoded in J2EE deployment descriptors.
- An implementation artifact (Java class, BPEL document, set of XSLT rules, etc) that defines the implementation of the component.

Finally, a *service component* (instance) represents a component actually deployed and accessible to other applications. It defined by the following.

- A component name.
- A service component implementation
- The values of any properties of the implementation that are being set to tailor the instance
- The specification of any services that resolve the "required" service specifications of the implementation. These may be "wires" that connect component instances or a "query" that executes to find a component at runtime that implements an interface, and has the right QoS policies and the required behavior (abstract process, etc.).

There are two basic approaches to defining a SOA component. The first is a *control file*: a document that, by reference, associates or joins all the parts of the component. For example, the control file may reference the WSDL definition (*interface provided*), the Java class that implements the component (*implementation artifact*), the associated policy documents (*policy assertions*), etc. The control file format gathers several individually-developed artifacts into a collection that, together, comprises the component. Application development tools aid in defining the control file.

The second format uses *pragmas:* structured comments (e.g. XDoclet tags [10]) or metadata language elements (as in JSR 175 [11]) specifying the same information, but contained within the body of a single source file. There is evolving support in Java [11] to make these annotations part of the language, but this approach does not

support other models like a set of SQL or XQuery statements. For example, structured comments in a Java source file indicate which Java methods will become Web service operations on the generated WSDL defining the component's service interfaces. We will illustrate this concept further in the discussion of individual component types.

2.1 Component Types

Because of the virtual nature of the SOA component model, many SOA components naturally support multiple implementation technologies. On the other hand, different implementation technologies are better suited for different tasks. To improve transparency, we introduced the notion of *service component types*, each suited for a developer with a given set of skills, performing a specific task, and using a certain tool. For queries, the programmer implements a .SQL file and a file containing a set of XQuery statements; for document conversion, XSLT style sheets, and so forth, using tools optimized for that task. There is no need to know that a Web service, Enterprise JavaBean (EJB) or other artifact is generated upon deployment, just that the overall result will be exposed and made available as a generic SOA component.

Programmers build a specific type of component adapted to the task, concentrating on the problem to be solved and the tool for doing so, not on the resulting artifacts. SOA development tools should focus on the skills of the developer and the concepts they understand. In the remainder of this section we take a brief look as some component types. When necessary, references are made to IBM products supporting the function being described.

Plain Old Java Object and Stateless SessionBeans
The most basic type of service component implementation is a "plain old Java object" (a "POJO"). JSR 109 defines the model and architecture for implementing Web services in J2EE [13, 14]. Tools like WebSphere Studio [15] can publish a Java class through a Web service abstraction. The Java class runs in the Web container, and has full access to the J2EE programming model's facilities. The WebSphere tools and runtime automate the conversion from SOA-encoded XML to the Java interface and operations of the POJO, and vice versa. Programmers may also use Stateless SessionBeans to implement services. WebSphere Studio tools automate publishing a Stateless SessionBean through a WSDL/SOA abstraction.

WebSphere Rapid Deployment [12] is a tool that simplifies defining a service in Java using the *pragma* format described previously. Using an editor, a programmer annotates the Java source file with control tags derived from the XDoclet model [10]. These tags specify whether the component is a POJO or Stateless SessionBean, the values for deployment descriptors (e.g. the transaction model), and the operations that must become part of the remote interface and WSDL.

IMS Transactions
The IMS SOAP Gateway [17] adds the ability to seamlessly expose existing and newly-created IMS application assets as Web services, in conjunction with the IMS Connect capabilities in IMS version 9. The gateway supports synchronous SOAP interactions over HTTP and HTTPS to enable the IMS application to receive inbound service requests. Additional functions such as SOAP client outbound support and

additional Web service protocols such as WS-Security, WS-Atomic Transaction, and WS-Addressing support are expected to be available in the near future.

The mapping of an IMS transaction to a Web service operation is defined by several files: an XML-COBOL converter, a WSDL Web service interface definition, and an XML correlator that relates the name of the application to the name of the XML-COBOL converter and provides protocol details for the connection between the SOAP runtime and IMS Connect. An XML Enablement utility in WebSphere Studio Enterprise Developer generates these artifacts to repurpose IMS COBOL applications as Web services.

SQL Statements

Products like the Websphere Information Integrator (WII) [18] enable databases to consume Web services. WII can make data sources described by XML schema accessible through standard SQL queries, the form familiar to DB2 programmers. The tools and runtime convert XML data sources to relational tables. A set of adapters provide a common WSDL-described interface for accessing XML information from WII. The basic SQL SELECT, UPDATE, and INSERT commands are integrated with compatible Web service operations. The DB2 database can invoke operations on Web services, both in queries and stored procedures, from SQL. Likewise, to publish enterprise information as Web services without programming, it is possible to expose SQL queries, database stored procedures, and other database artifacts as Web services.

3 Component Assembly and Customization

As has been mentioned before, composition is the core development task in a SOA programming model. We focus in this section on two forms of component composition that can be used to compose new services from existing ones. Each one derives from well established models of application integration and assembly.

1. *Structural composition* is the assembly of modules and solutions from existing components. Structural composition reflects the current practice of deploying solutions by assembling and connecting (logically "wiring" together) a set of existing components.
2. *Behavioral or process-oriented composition* describes the implementation of the composite service, called a *process*, via a classic procedural programming metaphor: what services to call, in what order, and how to aggregate the results. Process-oriented composition, directly derived from the legacy of workflow-oriented integration of applications [20] and human tasks, is one approach. Many programmers will also approach behavior composition through a state machine or {event, state, action} metaphor using, for example, UML State Diagrams [9].

Structural Composition

As we have seen, SOA components document the interfaces they need from other services (*imports*), and the interfaces they offer (*exports*). In structural composition,

programmers wire the required interfaces of a component to interfaces that other components or services provide. This *wiring* metaphor is similar to defining UML collaboration diagrams. The "wires" represent the flow of messages from one component's required interface to an interface that another component implements. Service composition can also connect a service's exported interfaces to event driven architecture (EDA) environments, allowing services' operations to be driven by subscriptions to events. Wiring can also connect imported interfaces, the interfaces the service calls, to topics to generate events that drive other services or software. The WS-Notification [31] family of specifications provides a model for integrate EDA with SOA.

A collection of services wired together into a bundle is called a *module*. A module can likewise declare imports and exports and be wired into a larger assembly, thus supporting a *recursive composition* model, so modules can aggregate other modules. Wires defined at assembly time are not satisfied until, at runtime, they are bound to deployed component instances.

An important concept in structural composition is that of *mediation* services. A mediation service defines the "behavior" of a wire, and is invoked by the SOA infrastructure (such as an Enterprise Service Bus (ESB) [21, 22]) whenever a message traverses the wire. Mediations typically do one of the following:

- Content based routing – Route the message to one or more alternative destinations based on content.
- Transformation – Transform messages and map operations, adapting the required interface to the implemented interface.
- Augmentation – Retrieve additional information to put the message into the form expected by the target service.
- Side effect – Perform an extra operation needed by the infrastructure or by an enterprise policy, beyond that specified in the data payload. For example, log financial messages.

Mediations are first class services, with supporting tools. WebSphere Business Integration Message Broker [23] for example supports powerful, complex mediations including augmentation, transformation and routing mediations.

Behavioral Composition

BPEL4WS provides Web services centric process composition. A BPEL4WS *process* is a directed graph[2] of *activity nodes* representing a single business activity—for example, a "quick loan" service in a banking business. Processes are classified as short- or long-running. Short-running processes have a single transaction per process and can be defined using basic process choreography. Long-running processes persist their execution state in a database. They require advanced process choreography and support transactionality at the activity level. They may include *compensations* to roll back partially completed work in the event of a failure, for long lived processes that

[2] BPEL4WS also supports other compound activities in addition to the directed graph model. For example, there is a language construct for a sequence of more basic activities.

cannot rely on the resource locking mechanisms of transaction managers, or for operations that lack transaction support.

A *business state machine* (BSM) is a service that aggregates other services and business logic relying on state based behavior. Consider the example of a purchase order processing service. The implementation of the *cancelPurchaseOrder* operation may depend on the "state" of the purchase order. If the purchase order has been entered, but not processed, there is one implementation of cancel. If, however, purchase ordering processing is complete and the PO is shipping, there may be a different implementation. A business state machine has one or more interfaces, which in turn have operations. The business state machine instance has a current state, and the state determines which operations are enabled.

4 Component Customization

A *customizable* component is one that can be tailored for reuse in a new context or within an assembly, or adapted to evolving business policies, without changing the source code. Our SOA programming model introduces two approaches to customization: adaptation, and mediation. *Adaptation* is achieved by providing *points of variability* (POV) on the component's behavior and its contract, which allow flexibility in the use of the component while not requiring any modification to the component's intrinsic implementation. The component provider declares points of variability by documenting a required interface. Other programmers configure or customize the component by providing a companion service that implements the POVs. The documentation of POVs is a generalization of the Strategy Pattern [24].

Mediation (selection) is a model in which the infrastructure or new customization logic processes in-flight messages. Processing can include routing to one of multiple implementations.

Consider an example of a commerce (shopping) component.

1. Discount algorithms change over time, and change from one organization to another. By declaring a POV *computeDiscount(shoppingCart),* the commerce component provider role allows another programmer to tailor the component's behavior over time, changing the discount computation without affecting the component's intrinsic behavior or source code.
2. A commerce component may require access to an inventory management service. The component provider cannot know which of several inventory systems a particular enterprise will use. By mediating the interaction between the commerce component's required interface and implementing services, it is possible to route and transform the messages for the proper inventory system.

5 Virtualization of Data Access

Service Data Objects (SDOs) [25] replace diverse data access models with a uniform abstraction for creating, retrieving, updating, iterating through and deleting business data used by service implementations. A SDOs *data graph* is a collection of

tree-structured objects that may be disconnected from the data source. Programmers use the single data graph abstraction to access data available through heterogeneous sources and technologies such as JDBC, the Java Messaging Service, Web services, Java 2 Connectors, RMI/IIOP, etc.

To maintain this abstraction, applications don't connect to a data source directly. Instead, they access an intermediary called a *data access service* (DAS) and receive a data graph in response. A DAS is an adapter that handles the technical details for a particular kind of data source. It transforms the data into a SDO graph for the client. To apply an update to the original data source, the application returns the updated graph to the DAS, which in turn interacts with the data source.

SDO sidesteps technology churn -- the rewriting of applications to keep up with shifting technology -- by encapsulating data access details to insulate business applications from technology changes. For example, consider a Java web application designed to read product descriptions from a database and display them as web pages. To access product descriptions in the database, the application might use JDBC heavily. Suppose that later the application topology changes, placing a web service between the application and the database. Now the application can no longer use JDBC to access the data and needs substantial rework to substitute a Web service data access application programming interface (API) such as DOM or JAX-RPC. SDO avoids this problem; an application written with SDO need not change.

6 Architectural Perspective

A runtime architecture supporting SOA and a SOA centric programming model comprises two broad categories of artifacts: service endpoints and the message transport fabric interconnecting them. A general architecture as provided by the IBM family of runtimes (none of which individually is the sole delivery vehicle for SOA) is illustrated in Figure 1.

At the core is an *enterprise service bus* (ESB) supplying connectivity among services. The ESB is multi-protocol, and supports point-to-point and publish-subscribe style communication between services, as well as being the container for *mediation services* that process messages in flight.

There are three key insights into the ESB concept:

1. WSDL and WS-I protocols provide the conceptual model. A specific deployed service may support additional optimized bindings, for example local calls or IIOP. Service implementers and callers are isolated from the optimizations.
2. There is a point-to-point, wire model for connecting component instances, but interfaces may also be connected to "topics" in a publish/subscribe infrastructure.
3. All calls may be mediated – The ESB is a logical concept, and may reside in endpoints when services are co-resident in a container.

A SOA component resides in an abstract hosting environment known as a *container* and provides a specific programming metaphor. The container loads the service's implementation code, provides connectivity to the ESB, and manages service instances. Figure 1 shows how different component types typically reside in different containers.

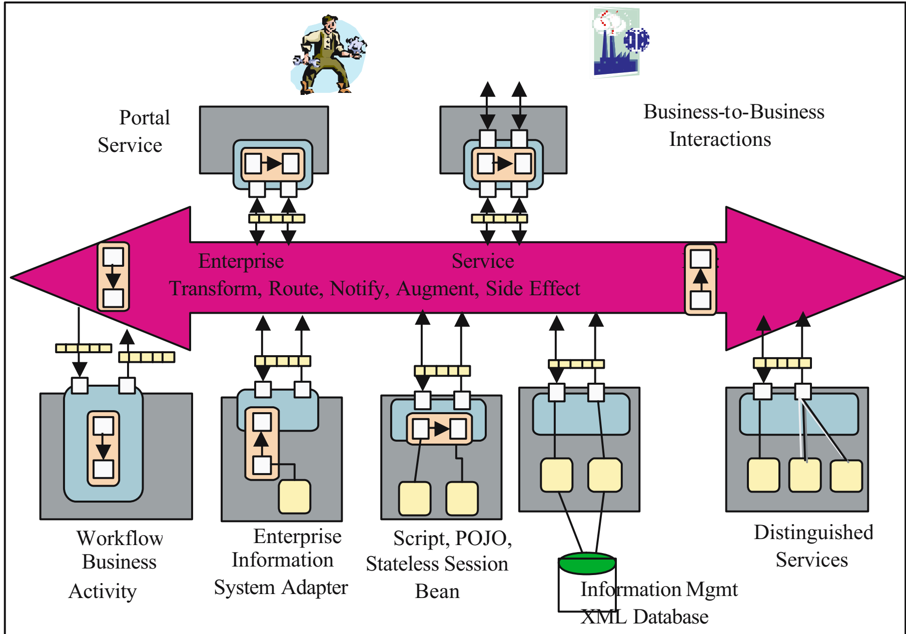

Fig. 1. A general service-oriented architecture

7 Related and Previous Work

Service Oriented Architectures are a paradigm or model which many enterprises have exploited for many years. The new concepts resonate with customers and have rapid adoption because they map to existing enterprise scenarios.

A key aspect of SOA is well-defined interfaces decoupled from implementation. There have been many previous systems employing this concept, most notably CORBA [26] and COM [27]. These approaches typically implemented an RPC model for connecting a caller to a component implementation, and supported a naming service for binding to a component by "name." J2EE [1] introduced a component model tightly linked with the Java language. J2EE uses Java interfaces for the IDL, supports declaring required interfaces through "ejbRefs" and "serviceRefs" and uses a naming service to bind required interfaces to implemented interfaces. An explicit role, the application assembler, manually connects the requested interfaces to deployed components that implement the interface. J2EE also supports component "policies" (deployment descriptors) for annotating a component with infrastructure requirements like security or transactions. Many of these concepts derive from IBM's Component Broker [29].

The SOA component model and SOA/Web services in general introduce several extensions or improvements to CORBA, J2EE, COM and other interface definition models:

- Contract languages (XML Schema, WSDL) are programming language agnostic. Even CORBA and COM IDL favored the C type space, while J2EE focuses on Java.
- XSD and WSDL are more tolerant of interface evolution. Element and operation reordering and addition typically do not affect implementations using prior versions.
- SOA and Web services inherently support both a call-return model and an asynchronous, one way messaging model. Previous systems either started with message driven processing and added an RPC model, or started with an RPC model and added asynchronous messaging. These approaches did not work well over complex, multi-hop, fire-walled Internet scenarios.
- SOA components support richer contracts that include quality of service properties and behavioral descriptions (using abstract processes) in addition to interface definitions. The SOA components model also introduces mediations and intermediaries.

Both J2EE and COM, and its evolution to .NET, provide some support for virtual access to data. J2EE introduced the concept of *container managed persistence* (CMP) for EntityBeans. COM and .NET also introduced the concept of ADOs [28], an abstraction for data that can map to multiple back-ends systems. SOA component models build on these approaches. The most noticeable improvement is linking the "data object" concept with the SOA model. There are two well-defined contracts in the SOA data object model: the contract between the component implementation and the data object (similar to ADOs and CMP EntityBeans), and a well-defined, data delivery and access service interface.

Finally, a key element of the SOA component model is the concept of "component types." J2EE introduced this concept with SessionBeans that implement the task model, and EntityBeans that represent the "data" model. The SOA component model we describe in this paper builds on this initial approach to introduce component kinds that more closely match the intent or tasks that programmers have when implementing a component/solution.

7.1 The Service Component Architecture

The Service Component Architecture (SCA) [30] is a common model for logic dealing with business data. SCA and Web Services together provide a framework for delivering SOA: SCA provides the common abstraction of implementation concerns and Web Services provides the common abstraction of interoperability concerns. SCA provides an implementation and assembly model for service oriented business applications. Here we briefly review the main concepts of the SCA programming model.

An SCA "implementation" provides the business logic for one or more services. Implementations can be written in many languages, such as Java, BPEL4WS, PHP, C, COBOL, etc. Implementations define their requirement on other services in form of "references". Further, an implementation can define "properties" that allow for configuration of its behavior. Both the services and references of an implementation are typed by interfaces. SCA is open with respect to the interface type system used (Java interfaces, WSDL portTypes, etc.) to type the services and references of the implementation, but favors the simple single-input single-output pattern standardized by WS-I [WSI] to promote interoperability among Web services.

Services, references, and properties define the configurable aspects of an SCA implementation, and together determine the "component type" of the implementation. An "SCA component" is defined in terms of a configured SCA implementation, by setting the values of the implementation properties and resolving its references to other SCA components via a "component wiring" specification. Finally, an "SCA module" is the packaging mechanism for implementations and components. Components are contained in the module assembly file that is part of the module package.

An SCA module can provide for the interaction between internal components and external applications by defining "external services" and "entry points". An external service allows components inside the module to access services outside of it; entry points are used to publish services of the module to external clients (outside of the module). External services and entry points use "SCA bindings" to configure the possible interaction mechanisms (Web services binding, stateless session EJB, etc.). SCA supports quality of service policies at the binding level and implementation level. Binding level policies are based on WS-Policy and define the quality of service (e.g. security, transactions, reliability, and so on) of the interaction across module boundaries. Implementation level policies are quality of service directives to the container hosting the implementation.

8 Conclusion

To support SOA requirements, a SOA programming model should support virtualization, multiple composition mechanisms, flexible component configuration, and loose coupling. The discussion of SOA programming models rises above the debate on the merits of different platform-specific technologies to a higher level of abstraction, integration and synthesis that is only achievable through the use of platform- and language-neutral standards. Standards are vital not only to insulate individual developers (who may not be IT professionals) from technology churn and enable them to utilize IT assets to perform their business duties. They are also vital to enable conceptual simplification by abstracting the alarming proliferation of software technologies, practices, tools, and platforms.

This article has described features of a new SOA programming model that can enable persons with different skill levels and different roles in the enterprise, not necessarily IT professionals, to create and use IT assets throughout every stage of the software development lifecycle. The result can be dramatically improved business agility for the on-demand enterprise.

References

1. Sun Microsystems, "Java 2 Platform, Enterprise Edition (J2EE)," java.sun.com/j2ee/1.4/download.html#platformspec.
2. Sun Microsystems, "Java Server Pages", http://java.sun.com/products/jsp/.
3. Sun Microsystems, "Java Servlets", http://java.sun.com/products/servlet/.
4. Sun Microsystems, "JavaBeans", http://java.sun.com/products/javabeans/.
5. "Web Services Description Language (WSDL) 1.1", http://www.w3.org/TR/wsdl, March 2001.

6. “Business Process Execution Language for Web Services (BPEL4WS) v1.1,” http://www.ibm.com/developerworks/library/ws-bpel/, May 2003.
7. “XSL Transformations (XSLT) Version 1.0”, http://www.w3.org/TR/xslt, November 1999.
8. R. Lerdorf and K. Tatroe, “Programming PHP”, O’Reilly, March 2002.
9. Object Management Group, “Universal Modeling Language 2.0 Superstructure FTF convenience document”, http://omg.org/cgi-bin/doc?ptc/2004-10-02, Oct 2004.
10. R. Hightower, “Enhance J2EE Component Reuse With XDoclets,” http://www-106.ibm.com/developerworks/edu/ws-dw-ws-j2x-i.html.
11. Sun Microsystems, “JSR 175: A Metadata Facility for the Java™ Programming Language”, http://www.jcp.org/en/jsr/detail?id=175.
12. IBM Corp., “WebSphere Application Server”, http://www-306.ibm.com/software/webservers/appserv/was/.
13. IBM Corp., “Build Interoperable Web Services with JSR-109”, http://www-106.ibm.com/developerworks/li brary/ws-jsrart/?ca=dnt-431, Aug 2003.
14. Sun Microsystems, “Java 2 Platform, Enterprise Edition (J2EE),” java.sun.com/j2ee/1.4/download.html#platformspec.
15. IBM Corp., “Websphere Studio”, http://www-306.ibm.com/software/info1/websphere/index.jsp?tab=products/studio.
16. S. Kim, “Java Web Start: Developing and Distributing Java Applications for the Client Side,” http://www-106.ibm.com/developerworks/java/library/j-webstart/.
17. IBM Corp., “IMS SOAP Gateway”, http://www-306.ibm.com/software/data/ims/soap/.
18. IBM Corp., “IBM DB2 Information Integrator Application Developer’s Guide v8.2”.
19. “XQuery 1.0: An XML Query Language,” W3C working draft, http://www.w3.org/TR/xquery/, February 2005.
20. F. Leymann and D. Roller, “Production Workflow. Concepts and Techniques”, Prentice Hall, September 1999.
21. R. Robinson, “Understand Enterprise Service Bus scenarios and solutions in Service-Oriented Architecture”, http://www-128.ibm.com/developerworks/webservices/library/ws-esbscen/index.html.
22. D. Chappell, “Enterprise Service Bus”, O’Reilly, June 2004.
23. IBM Corp. “WebSphere Business Integration Message Broker”, http://www-306.ibm.com/software/integration/wbimessagebroker/.
24. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, January 1995.
25. B. Portier and F. Budinsky, “Introduction to Service Data Objects: Next-generation data programming in the Java environment”, http://www-106.ibm.com/developerworks/java/library/j-sdo/, September 2004.
26. M. Henning and S. Vinoski, “Advanced CORBA(R) Programming with C++”, Addison Wesley, February 1999.
27. D. Box, “Essential COM”, Addison Wesley, December 1997.
28. D. Sceppa, “Microsoft ADO.NET (Core Reference)”, Microsoft Press, May 2002.
29. O. Gample, A. Gregor, S. B. Hassen, D. Johnson, W. Jonsson, D. Racioppo, H. Stöllinger, K. Washida, and L. Widengren, “Component Broker Connector Overview”, IBM ITSC, May 1997.
30. IBM Corp., “Websphere Integration Developer 6.0. Technical Product Overview”, available at http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/topic/com.ibm.wbit.help.prodovr.doc/pdf/prodovr.pdf.
31. WS-Notification”, http://www.ibm.com/developerworks/library/specification/ws-pubsub.