Service Component Architecture

Building Systems using a Service Oriented Architecture

A Joint Whitepaper by BEA, IBM, Interface21, IONA, Oracle, SAP, Siebel, Sybase.

Version 0.9

November 2005

Authors

Michael Beisiegel	IBM Corporation
Henning Blohm	SAP AG
Dave Booz	IBM Corporation
Jean-Jacques Dubray	SAP AG
Adrian Colyer	Interface21
Mike Edwards	IBM Corporation
Don Ferguson	IBM Corporation
Bill Flood	Sybase, Inc.
Mike Greenberg	IONA Technologies plc.
Dan Kearns	Siebel Systems
Jim Marino	BEA Systems, Inc
Jeff Mischkinsky	Oracle Corporation
Martin Nally	IBM Corporation
Greg Pavlik	Oracle Corporation
Mike Rowley	BEA Systems, Inc.
Ken Tam	BEA Systems, Inc.
Carl Trieloff	IONA Technologies plc.

Copyright Notice

Table of Contents

1 Introduction

Service Component Architecture (SCA) [1] is a specification which describes a model for building applications and systems using a Service Oriented Architecture (SOA). This white paper discusses the motivation behind SCA, describes the major features of the architecture and presents the areas for future development of the specification. The paper also explains how SCA extends and complements prior approaches to implementing services, and how SCA builds on open standards.

2 Motivation

SCA aims to simplify the creation and integration of business applications built using a Service Oriented Architecture (SOA). In an SOA, relatively coarse-grained business components are exposed as services, with well-defined interfaces and contracts. Interfaces are expressed using technology agnostic business terms and concepts. "Coarse grained" here means that the service interfaces use relatively few service methods to achieve a particular business goal, with large document-oriented parameters.

While SOA-based systems can have individual services that are built using object-oriented technology (among other approaches), the overall system design is service-oriented. In particular the service interfaces involve the exchange of business data, not the exchange of objects.

SCA also provides the capability to build coarse-grained service components as assemblies of fine-grained components. "Coarse-grained" means the use of interfaces with relatively few methods and where parameters and return values are typically document-oriented. "Fine grained" means that the interfaces may use a larger number of service methods, involving simpler parameter type.

SCA builds on emerging best practices of removing or abstracting middleware programming model dependencies from business logic. SCA aims to reduce or eliminate the "incidental" complexity to which application developers are exposed when they deal directly with middleware APIs. SCA allows developers to focus on writing business logic. However, SCA complies with existing standards "under the covers" to preserve existing investment in standards, middleware and tools. This approach is exemplified by a number of existing projects, such as the Spring framework [11].

The benefits of such an approach include:

- simplified business component development
- simplified assembly and deployment of business solutions built as networks of services
- increased agility and flexibility
- protection of business logic assets by shielding from low-level technology change
- improved testability

SCA is based on an open specification, allowing multiple vendors to implement support for SCA in their development tools and runtimes. This is particularly important for the deployment, administration, and configuration of SCA-based applications.

Unlike existing approaches such as Spring, SCA also supports a variety of component implementation and interface types as first class citizens. For example, the

implementation of an SCA component may be a BPEL process, and its interface may be defined in WSDL, or the component may be a Java class with an interface defined as a Java interface. This gives businesses the flexibility to incorporate a wide-range of existing and future assets into an SCA-based system with little or no bridging code required. It is this direct support for richer interface types that make SCA an ideal platform for delivering applications built using a Service Oriented Architecture (SOA) based approach.

2.1 Service Oriented Architecture and Web Services

SCA provides a first class model for building systems using Service Oriented Architecture (SOA). SOA is a composition model that connects the functional units of an application, called services, through well-defined interfaces and contracts between these services. SOA also emphasizes loose coupling between services. Loose coupling precludes undocumented interactions between services, for example through shared data, and it also supports the independent evolution of interfaces.

A service's interface is defined in a way that is independent of the hardware platform, the operating system, hosting middleware and the programming language used to implement the service. This allows services, built on a variety of systems, to interact with each other in a uniform and universal manner. In addition, the applications' interfaces and services are expressed using business terms and concepts – they are not technology focused.

The benefit of a loosely-coupled system is in its agility - its ability to accommodate changes in the structure and implementation of the internals of a service. By contrast, tight coupling means that the interfaces between the different components of an application are dependent on the form of implementation, making the system brittle when changes are made to components.

The agile nature of loosely-coupled systems serves the need of a business to adapt rapidly to changes in policies, business environment, product offerings, partnerships and regulatory requirements.

Service-oriented architectures are not new. Many organizations have built SOAs by using "best practices" applied to message oriented systems, RPC infrastructures, etc. SOAs are a more agile alternative to the more tightly-coupled object-oriented models that have been used to build distributed applications. While SOA-based systems may have individual services that are built using object-oriented technology, the overall system design is service-oriented. In particular the service interfaces involve the exchange of coarse-grained business data usually as documents in an interoperable format, not the exchange of objects.

SCA is a model designed for SOA, unlike existing systems that have been adapted to SOA. SCA enables encapsulating or "adapting" existing applications and data using an SOA abstraction. SCA builds on service encapsulation to take into account the unique needs associated with the ***assembly*** of netorks of heterogeneous services. SCA provides the means to compose assets, which have been implemented using a variety of technologies using SOA. The SCA composition becomes a service, which can be accessed and reused in a uniform manner. In addition, the composite service itself

can be composed with other services. SCA provides the ability to dynamically assemble these services to provide business capabilities, but it does so in a way that can be adapted and evolved as business requirements change and grow.

A principal technology for building SOA systems is Web services. Web services are mainly focused on two objectives:

1. The wire-level protocols that ensure runtime interoperability between heterogeneous systems. Web services are based on the exchange of messages using a technology-neutral XML format (SOAP).

2. Standards for defining service interfaces (WSDL and XML) independently of the implementation technology. A standard for defining interfaces also enables interoperability between tools.

SCA complements Web services, by providing a means of assembling services into a business system, as well as providing a service construction model. SCA takes advantage of many of the aspects of Web services. For example, SCA uses the capabilities of WS-Policy as a way of specifying a wide range of interoperability concerns including security.

SCA recognizes that using Web services aren't the only way to implement an SOA and so SCA supports a range of technologies. Service interfaces can be defined using WSDL and Java interfaces, and will expand to other interface languages. SCA service components can be built with a variety of technologies such as EJBs, Spring beans and CORBA components, and with programming languages including Java, PHP and C++. There are also evolving XML centric approaches to building services components. Some examples include BPEL [12], XSLT [15] and XQuery [16].

SCA components can also be connected by a variety of bindings such as WSDL/SOAP web services, Java™ Message Service (JMS) [13] for message-oriented middleware systems and J2EE™ Connector Architecture (JCA) [14]for enterprise information services. SCA recognizes that there is the need to utilize existing application components written using these technologies and also recognizes that some of these technologies have useful attributes and capabilities which are essential to business systems.

3 Architecture

SCA encourages an SOA organization of business application code, based on components that implement business logic, which offer their capabilities via service-oriented interfaces and consume functions offered by other components via service-oriented interfaces. This is illustrated in the following figure, which can be contrasted with the organization of existing applications:

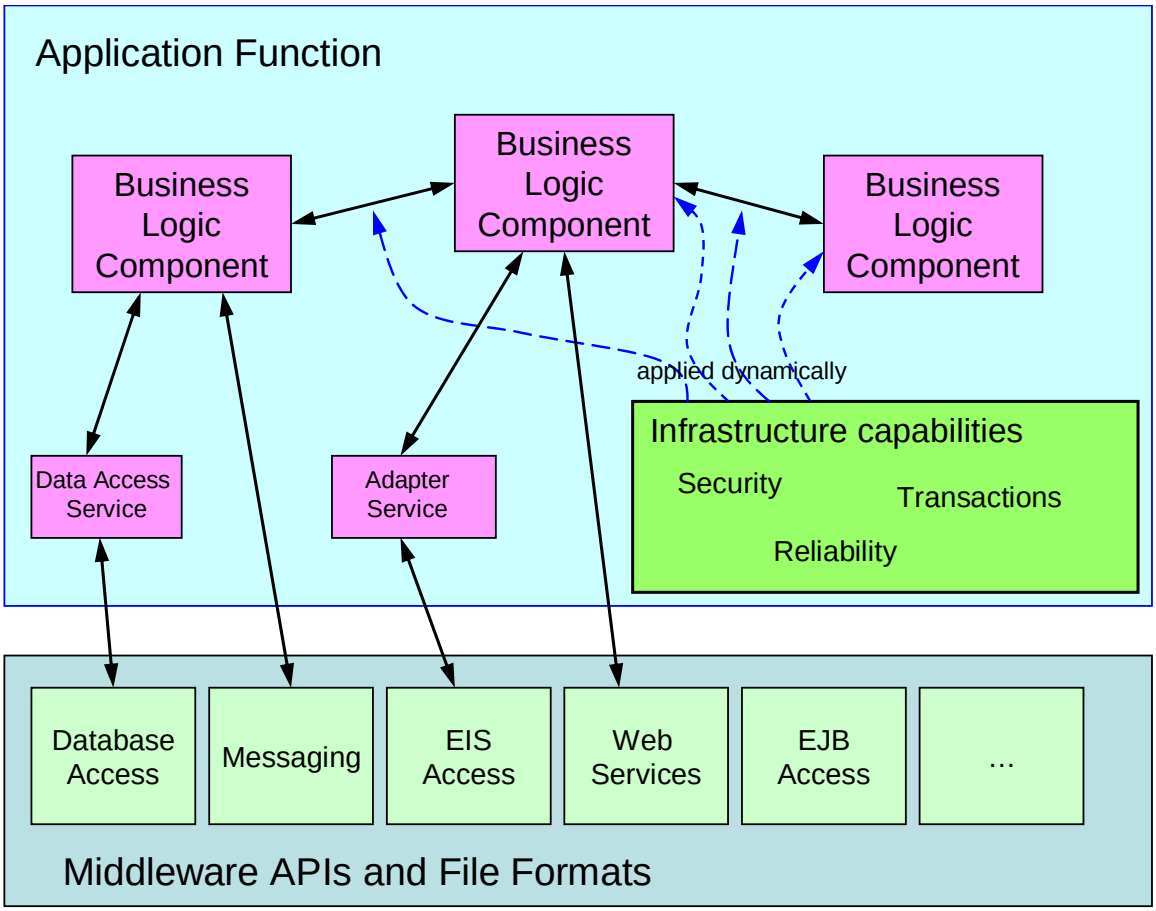

Figure 1: SCA Application Architecture

More advanced businesses already use an application architecture that is close to that shown in Figure 1, but the difficulty is that there is no industry-wide set of capabilities that provide support for this architecture. SCA aims to fill this gap and to help businesses create Service-oriented systems using common infrastructure and common skills.

SCA divides up the steps in the building of a Service Oriented Application into two major parts – first, the ***implementation*** of components which provide services and which consume other services; second, the ***assembly*** of components to build the business application through the wiring of service references to services. SCA also provides a means of packaging and deploying sets of closely related components which are developed and deployed together as a unit.

The model in figure 2 also decouples service implementation and assembly from the details of infrastructure capabilities and from the mechanisms for invoking external

systems. This enables ***portability*** of services between different infrastructures. This portability, building on the portability of implementation technologies like Java and BPEL4WS, complements the runtime and tool interoperability of Web service standards.

3.1 Service Implementations and Service Clients

Service implementations are units of business logic, written using any one of many implementation programming languages, including conventional object-oriented and procedural languages such as Java, PHP, C++, COBOL and C. It is also possible to implement services using more XML centric languages such as BPEL and XSLT transformations, and declarative languages like SQL and XQuery. The freedom to use the most appropriate implementation type is an important aspect of SCA – the implementation is the servant of the business process, not the other way round.

To assist in understanding how SCA implementations are written and assembled, there is an example in the document "Building your first application – Simplified BigBank" [3] – it may be useful to refer to this as you read the following sections. In the following sections, this example is called "BigBank" for short.

An implementation can provide a ***service*** – which is a set of operations defined by an ***interface*** that can be used by other components. Implementations can also use other services – these are service references (***references*** for short) which indicate a dependency that the implementation has on services provided elsewhere.

In the BigBank example, there is a service called AccountService which provides full account information for a customer and which is implemented by the AccountServiceComponent. The AccountServiceComponent in turn uses references to the AccountDataService and to the StockQuoteService. References are defined by interfaces. References provide a level of indirection between the implementation and the target service, since the actual target service used is configured by the assembly and it could be bound dynamically at runtime.

An implementation may also have one or more ***properties***. A property is a data value that can be configured externally and which can affect the business function of the implementation.

The BigBank AccountServiceImpl implementation of the AccountService has a property called ***currency*** which is used to set the currency that the account service uses when preparing account information.

Service implementations may be designed to provide coarse-grained, remotable services intended for use remotely by clients (eg in another department or in another business). Coarse grained services typically use document-style business data for parameters and return values, and it is recommended that these parameters are represented as Service Data Objects (SDOs) (see the SDO Specification [2] for more detail about SDO). Other service implementations may be designed using a fine-grained service interface intended for local use by clients within the same process. For local interfaces, a larger set of parameter types is expected.

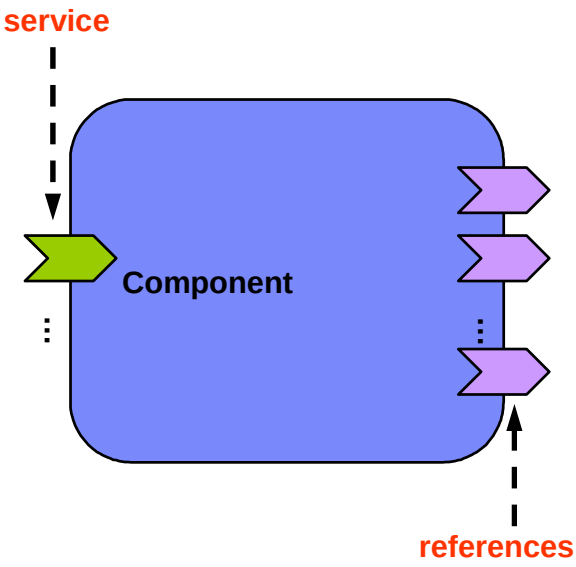

Figure 2: An SCA Component

The service, the references and the properties of an implementation can be thought of as the externally configurable aspects of the implementation. An important idea associated with this "outside" view is that it can represent a boundary between the programmer responsible for building service implementations and a service assembler responsible for assembling the services into an overall business solution. All components have the same external shape or model to the assembler, independent of their internal implementation languages and design.

While the service offered by an implementation is fixed, both the references and the properties can be configured, to produce a ***component***. Configuration of a reference involves binding the reference to a target service, which will then be used by the implementation when it invokes the reference. Configuration of the properties involves setting specific data values for the properties. This means that one implementation can be used to build multiple different components, with each component having a different configuration of the references and properties.

In the case of BigBank, the AccountServiceComponent configures the AccountServiceImpl implementation, wiring its references to the AccountDataServiceComponent and to the StockQuoteService and setting its currency property to "EURO".

Components and their service interfaces can be designed for purely local use by other components, or components can be designed for remote access, either from other parts of the business or from other businesses. Local components can use interfaces optimized to exploit the co-location of the service client and the service implementation. Services designed for remote use must take account of the potential for the client being connected over a remote link and so must offer interfaces that are compatible with this remoteness.

Dependency injection, as supported by containers such as Spring, has proven to be a valuable technique for implementing business applications and hiding details of

service location and instantiation. SCA applies the technique of dependency injection to configure SCA services, using it to provide a component instance with its references and property values. Injection can be seen in operation in the example code [3]. SCA does provide a minimal set of APIs for situations where the service implementer is not able to use dependency injection.

Programming a service invocation is done consistently, without regard for the access mechanism used to communicate with the target service – which is invisible to the business logic. This allows for the access mechanism to change over time, or for multiple access mechanisms to be used depending on the client and provider involved, without the need to modify the service implementation or the client invocation. For example, a single service implementation could support access via Web services and also support access via a JMS-based transport, without requiring service clients to change code.

3.1.1 Java Implementation Type

Java is a first class implementation type for SCA: `<implementation.java>`.

The aim of SCA is to support the implementation of components using Plain Old Java Objects (POJOs), requiring no special programming or APIs in order to implement the business function of the component. In addition, the separation of the business interfaces from the actual target services used to satisfy references allows for the assembly step to re-target the reference at will.

In addition to simple Java classes, SCA anticipates that there will be more specialized forms of Java implementation, which can take advantage of more sophisticated runtime support. An example is the use of an Enterprise Java Bean (EJB) as an SCA component – this would have the implementation type, `<implementation.ejb>`. In this case, the implementation code must conform to the specification for EJBs, but the implementation can take advantage of the capabilities provided by an EJB container, such as the persistence model of EJB 3.0. See the sections on J2EE 1.4 Integration and Java Enterprise Edition 5 for more details.

3.1.2 BPEL Implementation Type

The OASIS WS-BPEL standard [12] is frequently used as the standard process language of providing business process integration within SOA implementations. BPEL is a native SCA implementation technology, with components implemented in the BPEL language being handled with a first class implementation type: `<implementation.bpel>`. SCA is a natural complement to BPEL components, since the BPEL processes typically coordinate the activities of other services, for which SCA can provide the wiring. SCA can also be used to publish the business process embodied in a BPEL process, making it accessible as a service for other SCA clients to invoke, without the need for the clients to be concerned about the details of how a BPEL process is invoked.

As BPEL processes are frequently asynchronous in nature, asynchronous BPEL processes will be able to take advantage of the native asynchronous capabilities within SCA, such as callbacks and conversational services. Until SCA, seamless callback integration between BPEL and non-BPEL processes was a proprietary exercise within

most vendor implementations. SCA provides a standard model for BPEL and non-BPEL interactions.

BPEL processes can also take advantage of the ability of SCA to attach infrastructure services to components, such as Security and Transactions.

3.2 Assembly

Assembly is the process of composing business applications by configuring and connecting components that provide service implementations. SCA assembly operates at two levels – the assembly of loosely connected components within a System and the assembly of closely connected components within a Module. These two levels roughly correspond to "programming in the large", creating business solutions as networks of loosely coupled services working together and "programming in the small", assembling services from closely related fine-grained components.

3.2.1 Module Assembly

An SCA ***Module*** typically contains one or more components that are deployed together into an SCA system. Services that are offered for use by components outside the module are represented as ***EntryPoints*** in the module. Where components in the module depend on services provided outside the module, the dependencies are represented as ***ExternalServices*** in the module. Viewed from the outside, the resulting module has services and references that correspond to the entry points and external services that are in the module

In the BigBank example, the AccountServiceComponent is part of the accountmodule module, which also contains the AccountDataServiceComponent, an entry point called AccountService which offers the account service to external users, and an external service called StockQuoteService which may reference a remote web service which provides the StockQuote function.

A module is assembled by configuring and wiring together components, entry points and external services. A very simple example of a module is shown in Figure 3:

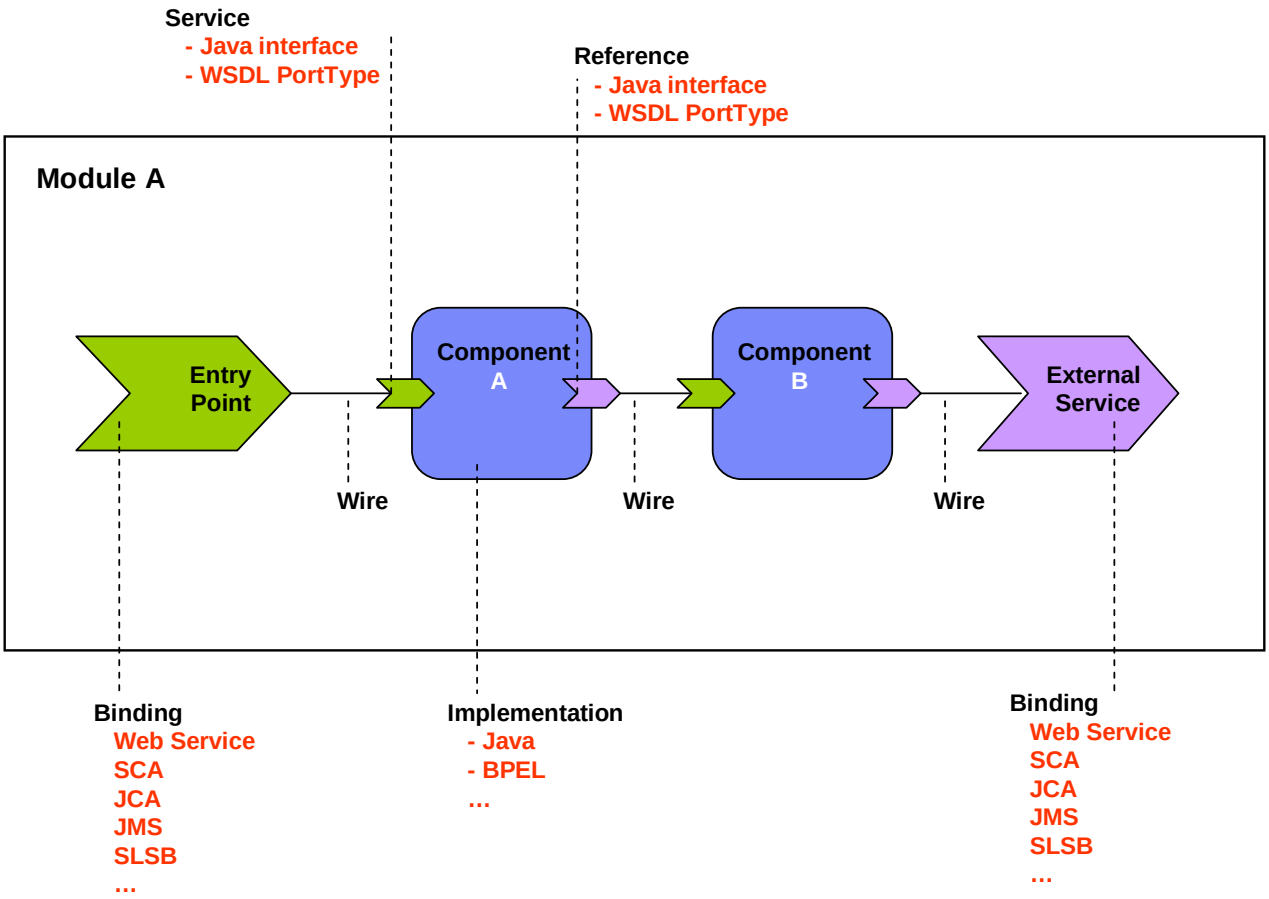

Figure 3: Example of a Simple Module

In this module, there are two components, called Component A and Component B. Each component provides a service and each has a reference to a service it depends on. The service provided by component A is made available for use by clients outside the module by the entry point and the entry point is wired to the service of the component. The service required by component A is provided by component B. The service required by component B exists outside the module and so there is one external service in the module, with the reference of component B being wired to the external service.

Where there are multiple components in a module, as here, some component services and references may be wired to each other entirely within the module. These are "local" services and the boundary of the module means that they cannot be seen or used outside the module. Only services and references explicitly exposed by entry points and external services can be seen and used outside the module.

3.2.2 System Assembly

Assembly at the ***System*** level represents the creation of a business solution through the configuration and wiring together of loosely coupled services.

The assembly of an SCA system mirrors the assembly of a module. The system consists of one or more ***ModuleComponents*** and potentially ***EntryPoints*** and ***ExternalServices***, plus the ***Wires*** that connect them. Module components represent configured instances of an SCA module, where the module component can set values for the external services of the module and can set values for properties exposed by the module. The system may contain an entry point for a service that is offered for use externally – for example, an entry point can be used to make a service available for use by other organizations or by customers as a Web service. The system may contain an external service for any service referenced by a module component in the system that is supplied by another organization – for example, a Web service provided by another business.

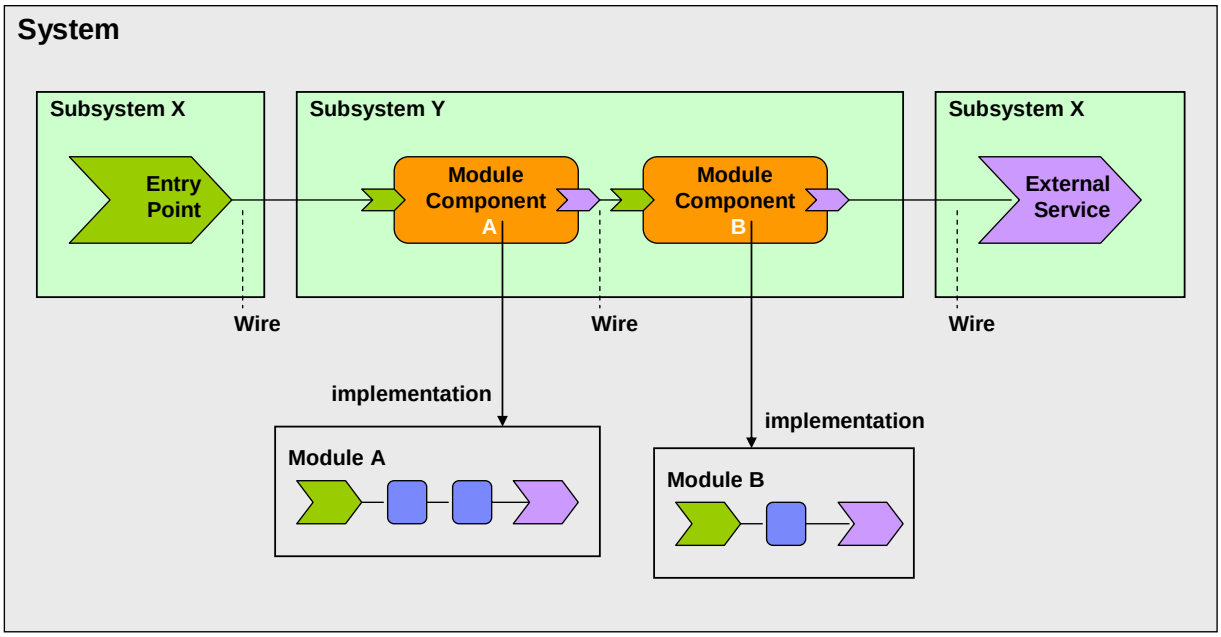

Figure 4: Example of System Assembly

For convenience, the configuration of an SCA system can be divided up into a series of smaller sections, each of which can be updated and deployed separately. These smaller sections of a system are called ***Subsystems***. The configuration of the whole system then consists of combining together the configuration of each of the subsystems. A subsystem can consist of any of the elements of a system. In particular there can be subsystems that consist entirely of wires which connect services and references defined in other subsystems. This allows the wiring to be updated independently.

The extent of an SCA system is flexible and can vary from business to business. At the smallest, a system could consist of a set of modules deployed and running on a single server. At the largest, a system could consist of all the information technology assets of the business, spanning multiple machines of differing types in many different locations. One of the important aspects of the SCA system is the concept of administrative control – the system represents the extent of control of an organization. Whether a single business has just one SCA system or uses multiple systems (e.g. one

per department) is a decision that is related to the size and organization of the business itself.

3.3 Bindings

Bindings are used to define the access mechanism used when a service is invoked remotely, such as Web services, JMS/Messaging, CORBA IIOP and Database Stored Procedures. Bindings are used by entry points and by external services. For an entry point, the binding defines the access mechanism which clients must use in order to access the service offered by the entry point. For an external service, the binding describes the access mechanism that is used when the remote service is called.

Bindings are configured and managed independently of the implementation code. The separation of the access mechanism from the business logic in the implementation code is a key aspect of SCA, which permits the same business logic to be used with a variety of different access mechanisms and which permits a system assembler to change and add access mechanisms as the needs of the business evolve.

SCA supports a series of different binding types. Examples include Web service, JMS messaging, stateless session EJB, data base stored procedure, EIS service. SCA also supports a non-interoperable binding, called the SCA binding, which allows the SCA runtime implementation to provide a highly optimized transport and protocol. SCA is extensible and additional bindings can be added if required.

Services called locally, that is, between two components of the same module, do not require bindings. Local invocations occur within the same process and involve calling from the client to the provider by means that are native to the implementation code, such as Java method invocation.

3.4 Asynchronous and Message-Oriented Model

The handling of asynchronous styles of programming is important for SOA solutions, since there are numerous application patterns where a simple synchronous call-and-return style is not appropriate.

SCA defines a set of facilities in support of a number of styles of asynchronous service invocations. The simplest form is where a service is invoked “one way” in a non-blocking style – the service is invoked and the client carries on executing without waiting for the service to execute. The “one way” style implies that the client does not expect to receive any data back from the called service. This can also be described as “sending a message” from the client to the service provider.

A more sophisticated form of asynchronous invocation is where the client provides a ***callback*** interface to the service that it invokes. The callback interface becomes part of the contract between the client and the service – effectively the services are bidirectional and imply that calls are made in both directions between the client and the provider. The timing of callback invocations is asynchronous – they may occur any time after the initial service invocation is made.

The BigBank example is a simplified one and does not use asynchronous programming. However, an example of where asynchronous service invocation might

be useful in the BigBank example is for the StockQuoteService. This service might take some time to complete, in which case the AccountServiceImpl could be written to invoke the StockQuoteService asynchronously, passing a reference to a callback interface. The AccountServiceImpl would implement the callback interface and when the StockQuoteService completed its calculation of stock values, it would pass back the result by calling the callback service.

Finally, SCA provides support for ***conversational*** services, where a series of interactions may occur over time between a service client and the service provider. The BigBank application might use this form of service when processing a loan application, where a LoanApproval service might require a number of different pieces of information from the AccountServiceComponent in order to complete its processing, but where the actual information required could vary depending on the status of the bank account. In this case, the LoanApproval service would require the AccountServiceComponent to supply a conversational interface with a series of services for the different types of information required, and would use them in sequence in order to complete the loan approval process.

Note that the *reliability* of message delivery is a property of the binding used, and is transparent to both the sending and receiving service. This becomes significant in the case of one-way style of invocation, where there is no straightforward way for the sender of a message to know whether it was received by the target service.

3.5 Infrastructure Capabilities and Policies

There are sets of attributes that may be required when calling a service from a client, which go under the collective heading of “Infrastructure Capabilities”. Examples of infrastructure capabilities include security attributes such as authentication and encryption requirements, transaction characteristics and whether reliable delivery of messages from client to service is required.

SCA models infrastructure capabilities in a declarative fashion. This ensures that these aspects of the overall system are separated from the business logic and that they can be modified without the need to change code, since there is no infrastructure code included in the business logic of an implementation. This simplifies the development of the business logic.

SCA defines infrastructure capabilities through the use of ***Policies***, which are bundled collections of attributes and settings. SCA attaches policy definitions to components, to entry points and to external services.

Some infrastructure capabilities can be quite complex, particularly security. An important goal of SCA is to hide as much of the complexity as possible from application developers and assemblers, while allowing experts in security and other policy areas to specify the policies that must be used with appropriate detail.

SCA simplifies the potential complexity of infrastructure policy for quality of service through the use of high level policy profiles. For example, an organization may have a small number of pre-defined policies that specify different levels of security required for different types of service and/or assets within the organization. Each policy may

describe settings for a set of policy attributes, such as the encryption of messages, authorization roles, authentication methods and so on. For a particular service, one of these pre-defined policy sets is applied by referencing the policy by name. This simplifies the application of infrastructure policy, ensures that the combination of policy settings is valid and facilitates quality assurance and validation.

The policy may require the configuration of one or more parameters to enable it to be used at runtime – for example, a security policy might need the name of a role required in order to use a particular service. The setting of parameters of this type is handled in the configuration of the component concerned.

SCA allows for policies to be aggregated into wider policies. For example, security, transaction and reliable messaging may each have its own policy set defined at a low level, but for general usage, the individual policies are aggregated into a smaller set of general policies that define all aspects in a single package. The general policies are the ones that are used in configuring components, entry points and external services. This further improves simplicity and the reliable use of policy.

SCA conforms to the WS-Policy [17] standard for policy definitions, while providing straightforward means to integrate policies with the assembly of SCA modules and systems.

3.6 Extensibility of SCA

SCA aims to accommodate a wide variety of technologies. As a result of this, SCA recognizes that it would be difficult to handle all potential technologies using a fixed set of features. SCA provides extensibility mechanisms which allows SCA to be extended to integrate technologies not described in the main specification.

There are 3 principal places for extension:

1. Interface types
2. Binding types
3. Implementation types

Interface types deal with technologies used to define service interfaces. SCA describes Java interfaces and WSDL as two technologies in this space. Other technologies for defining interfaces can be used, particularly where an implementation type naturally uses a different method for defining its interfaces.

Binding types deal with different technologies for accessing a service. There are potentially many of these and it will not always be desirable to map them to Web services, for example. SCA allows for additional binding types to be defined, although new bindings will require the SCA runtime to have additional support added to use these new bindings. Adding the support for new bindings is transparent to application components. For example, adding a binding that supports the Session Initiation Protocol or SMTP requires extending the SCA runtime, and perhaps defining new policies, but would be transparent to application components.

Implementation type describes the technology used to build a component implementation. In principle this can be a very large set. In addition to programming technologies, such as Java, COBOL, C++, BPEL and PHP, there are other types of implementation that are very useful in the services world. Examples include declarative styles of programming such as XSLT scripts, XQuery and SQL. There are also container-based solutions such as EJBs, Spring beans and Corba components. SCA is extensible so that all these different forms of implementation could be accommodated in an SCA runtime.

In some cases, when considering how to link an existing technology to SCA there may be a choice about whether to define a new binding type or to define a new implementation type. When a service runs within a runtime that does not support any SCA capabilities, then the component should be accessed using a binding that is appropriate for communicating with the component's runtime. However, if possible, the service runtime can be extended with SCA capabilities and a new SCA implementation type can be created for that runtime.

The SCA abstract model is designed to ensure that implementing an SCA runtime with support for native bindings, interfaces and implementation types is natural. It is anticipated that there will be SCA compliant runtimes for relational database systems, transaction processing monitors, and so on. Web service protocols, along with optimized bindings and protocols, will provide interoperability between SCA runtimes.

3.7 In Summary: SCA Model Characteristics

1. Application logic is divided up into application components that implement business services
2. Components have business-oriented, service-oriented interfaces. Components do not have interfaces that reflect middleware abstractions; they have interfaces that reflect business abstractions
3. Application components can be reused by "wiring" together new and existing application components to create new solutions. This assembly capability of SCA can integrate existing and new assets based on multiple heterogeneous technologies into a composite service network.
4. SCA implements a separation between the concerns of a component implementer and the concerns of a system assembler creating a solution by wiring together existing components and services.
5. SCA can be implemented on top of a broad range of middleware environments
6. Components are described and used in the same way regardless of the language or technologies used to implement the component.
7. SCA allows "qualities of service" such as transactions, security and reliable asynchronous invocation to be applied to components declaratively and dynamically without requiring programming using complex API calls.

8. Irrespective of whether a component is local to the deployment unit or remote, the component is accessed through its defined business interface. SCA provides for assembly of components at multiple levels, allowing greater control and visibility of application artifacts.
9. A variety of resources such as Web services, EIS functions, remote EJBs can be modeled as remote components, and can generally be used without regard to the underlying implementation technology or of the transport. Some transports impose limitations on the qualities of service that can be supported.
10. SCA supports multiple technologies for expressing the interfaces of components, including WSDL and Java interfaces.
11. Components with business service interfaces are used to provide access to data – separating issues related to data persistence from the business logic. This also facilitates portability of components between different runtimes.
12. Infrastructure capabilities, such as Security and Transactions, are applied to component interactions rather than being accessed through code. This helps keep the business logic code clear of infrastructure concerns.
13. Application components can be “customized” either at development time or at run-time, by delegating decisions to other components using the “strategy pattern” [18].
14. SCA defines an abstract model for implementing components and for accessing components. The model supports multiple concrete implementations in a wide variety of programming languages and technologies, including Java, C++, BPEL and XSLT scripts. SCA attempts to be "minimally intrusive" with few APIs and, where supported, uses techniques such as dependency injection to eliminate the use of APIs altogether.
15. The preferred form for data exchanged between components via remotable business interfaces is that defined by the Service Data Objects (SDO) specification [2].

4 Use Cases

SCA aims to support potentially large and complex business solutions built using SOA. However, there are some typical use cases for SCA that are used as the building blocks of larger applications. This section describes some of these use cases.

4.1 Providing a Web Service to External Users

Providing a Web service which can be used by external users (for example, users in other businesses, connecting via the Internet), SCA involves the following basic steps:

- Define the service interface, concentrating on the set of business operations required and the format of the data (message) that is supplied to each operation and the format of the data that is returned. The service interface can be defined using WSDL or with a Java interface.
- Write an implementation of the service, using one of the implementation types, such as Java or BPEL, providing the business function for each operation.
- Create an SCA module which contains a component which configures the implementation. The module has an entry point which declares the service interface and which has a Web services binding – this makes the service available for external users. The entry point references the component.

4.2 Connecting to an External Web Service

To use a Web service provided externally involves:

- Obtain the WSDL which defines the Web service interface.
- (optional, if producing a Java implementation) Produce a Java interface that is a mapping of the WSDL interface. Using a Java interface makes the programming of a Java implementation simpler.
- Write an implementation which uses the service, providing the business function which uses the Web service. This could be done using Java or using BPEL, for example. Declare a reference to the target web service.
  In Java, the implementation code is given the service through dependency injection (or through an API, when dependency injection isn't available) and it concentrates on the business function and the data values which are passed to the service and returned from it.
  In BPEL, the reference service is simply a partner link which is satisfied by the actual service wired to the reference in the SCA configuration. The partner link is used when the BPEL process executes.
  In all implementations, no details of the Web services protocol are exposed to the implementation code.
- Create an SCA module which contains a component which uses the implementation. The module also has an external service which declares the service interface and which has a Web services binding pointing to the target service. The reference of the component is wired to the external service.

4.3 Asynchronous Use Case

Asynchronous services are a common pattern for applications built using SOA. A simple example often used is the case of a travel agency booking a trip for someone, where the trip can involve booking a series of items from different suppliers – for example, a flight, a rental car and a hotel. The reason for doing this is that the booking with each supplier can take some time and it will save time overall if the three bookings are executed at the same time, rather than performing them one after the other.

A way of writing the travel agency application is where the steps involving the booking of the flight, the rental car and the hotel are all modeled as asynchronous services. This can be done using a callback style of interaction, for example, where each of the booking services calls back to the travel agency application once it is complete. The travel agency application is then set up to call all three services and to wait for the callbacks to arrive from each of the booking services – or to time out if all three do not respond within some set interval.

It is common to write a business process, such as the travel agency trip booking process in this example, using the WSBPEL language [12]. WSBPEL makes the use of asynchronous services a particularly natural way of building business solutions. SCA makes it straightforward to create components that use asynchrony and to assemble them into an overall business solution. In this case, each of the asynchronous services (flight booking, etc) could use a callback interface to signal their completion to the trip booking process. This would be reflected in a series of partner links within a BPEL implementation of the trip booking process.

5 Integration with Commonly Used Technologies

Frequently when assembling new business processes and composite applications, it is most cost effective to incorporate existing infrastructures into the new development, rather than altering or replacing them. On other occasions, there are performance requirements that can not be achieved unless a native binding to the underlying endpoint or implementation is used. For example, there is a significant set of existing EJB based applications across the industry.

SCA provides the ability to access services that are deployed outside of the SCA system through the use of external services, which may use any of a variety of bindings for communicating with the service provider. Similarly, entry points are available for making SCA services available to external consumers via a variety of bindings. This binding approach is the simplest form of integration of SCA components with code implemented using other commonly used technologies.

However, it is frequently desirable to mix SCA services developed using the SCA programming model with code developed using other technologies. When used in this way, SCA provides a unifying mechanism for wiring service clients to service providers irrespective of the technologies they were developed with. Many application framework technologies offer similar sets of fundamental capabilities, such as publishing services, using services, and configuring services. These technologies can be mapped onto the equivalent concepts within SCA.

For each technology, there is a set of possible techniques for integration - call the technology being integrated “X”:

1. Call from SCA to X and from X to SCA using a Binding.
2. X concepts are configured via SCA module components (X defines an entire SCA module) and SCA system assembly can be used to integrate the module component with other module components.
3. Create a new ***implementation type*** that allows hosting X component implementations as SCA components (X provides pieces of an SCA module)
4. Develop SCA module artifacts as part of the implementation of X (e.g. an SCA module as part of a J2EE WAR)

The current SCA assembly specification includes general mechanisms for integrating other technologies, but detailed descriptions of specific integrations do not yet exist. The rest of this section provides thoughts on the approach that will be taken for some significant technologies.

5.1 Using the SCA Client Model

Whenever code based on another technology is used as part of an SCA module, the non-SCA code can use SCA services via the SCA client model. In the case of Java, any code that is in the same module as SCA components can get access to an SCA service with code that looks like this:

```
ModuleContext mc = CurrentModuleContext.getContext();
StockQuoteService sq =
  (StockQuoteService)mc.locateService(“StockComponent/QuoteService”);
```

```
float price = sq.getQuote( "AZZ" );
```

When services are accessed from non-SCA component code, the services are located by the name of the service within the module, rather than by configuring and using an SCA reference.

When SCA is integrated with technologies which support dependency injection, like Spring and EJB 3.0, the reference to the SCA service could be injected (it would be present in the configuration of the technology), so no API call is necessary. This is not possible for technologies like EJB 2.1, which require the explicit use of APIs.

5.2 J2EE 1.4 Integration

SCA provides specific support for J2EE applications and component types to make the inclusion of existing J2EE assets into an SCA system as simple as possible. J2EE 1.4 artifacts can be integrated with SCA at a number of different levels. J2EE applications can be used in their entirety as a module within an SCA system. This is called system-level integration, since the communication is between components at the system level.

It is also possible to use J2EE modules within SCA modules, which is called module-level integration. With this level of integration, an SCA module may reference components whose component type is defined using a J2EE module (WAR, or EJB JAR). This closer level of integration can also involve the direct use of EJBs as SCA implementation types.

For both system-level integration and module-level integration, it is possible to map J2EE environment variables onto SCA properties. The name of the property is the same as the JNDI name of the environment entry, without the `java:comp/env` prefix. For system-level integration the property will have already been given a value by the J2EE application, but SCA can override this by specifying a property value on the module component within the system.

The immediate value of this first class integration of J2EE with SCA is in enabling the participation of the large installed base of J2EE 1.3 and 1.4 application environments which include:

1. Custom applications built with Servlet, JSP, EJB and JMS foundation technologies
2. Third party JCA adapters, frequently provided by third party adapter vendors, for business applications such as SAP, Oracle, Peoplesoft, Siebel and others.
3. Integration products based on J2EE 1.3 and 1.4 technologies

5.2.1 Using SCA Services from Servlets and JSPs

Where a Servlet or a JSP needs to access SCA services, it uses the SCA client model defined previously.

5.2.2 Linking to J2EE Applications via Bindings

J2EE applications which expose services as Web services can be accessed from SCA using a Web services binding. Where J2EE applications expose Stateless Session Bean interfaces, a SCA provides a native binding which can be used to invoke the EJB interfaces from an SCA client using the RMI/IIOP protocol. Equally, an SCA component can expose its service using the same binding – allowing that component to be invoked from an EJB as if it were an EJB reference.

5.2.3 System-Level J2EE Integration

Using a J2EE application deployed within an SCA system as a complete SCA module, integration between SCA and J2EE stateless session beans is possible. Each remote stateless EJB within a J2EE application is mapped to an SCA entry point whose interface is the remote interface of the EJB. An EJB can invoke an SCA service as if it were an EJB if the SCA service offers an EJB binding, with the EJB reference being treated as an SCA external service.

System level integration of a J2EE application is also possible where it provides or uses J2EE Web services. Web services exposed by the J2EE application are mapped to SCA entry points and Web services used by the J2EE application are mapped to external services, both using Web services bindings.

5.2.4 Module-Level Integration with J2EE

Closer integration between SCA components and J2EE is accomplished by using stateless EJBs as components within an SCA module. There is an implementation type that corresponds to this: `<implementation.ejb>`.

When an EJB is used as a component type, the service provided by the implementation is the same as the remote interface of the EJB. The SCA properties are the environment entries defined for the EJB. An EJB may have required EJB references that are not resolved within the EJB J2EE module: these are treated as service references of the component and they can be satisfied by wiring them to SCA services offering an EJB binding.

5.2.5 JCA Adapters

JCA resource adapters provide a mechanism within J2EE for connecting to enterprise applications. While J2EE applications that make use of resource adapters can still work within an SCA system, it is not expected that SCA components will directly use standard JCA resource adapters. One way for an SCA component to make use of an enterprise application is through an external service binding that uses a protocol that is provided by the enterprise application. An alternative way that SCA components might communicate with a JCA adapter is via a stateless session EJB, which, in turn, communicates via the adapter.

Newer JCA adapters may implement the interfaces defined by the Enterprise Metadata Specification (EMD) [8]. Services in the enterprise application can be discovered (via tooling) using a JCA adapter which supports EMD. The discovered service, an EMD Service Description, can be saved as an SCA entry point or as an SCA external service with a JCA binding.

5.3 Spring

The Spring Framework [11] is a popular platform used to construct Java applications. It aims to reduce the complexity of the programming environment and shares many of the same design principles as SCA. In particular, Spring provides a runtime container that provides dependency injection so that application components can avoid the need to program directly to middleware APIs.

SCA views Spring as a natural partner which can be used as a component implementation technology. The Spring framework can be used to create components and wire them within a module using its dependency injection capabilities. SCA may be used to extend the capabilities of Spring components by publishing Spring beans as entry points to be accessed as services by other modules as well as by providing Spring beans with service references wired to services of other modules. As the SCA specification evolves in the community, it is hoped that SCA and Spring will define a deeper integration so that developers can further leverage the strengths of both technologies in their applications.

SCA can add useful capabilities to an application implemented using Spring, for example:

- support for remote components and multiple protocols
- support for components written in a variety of programming languages
- support an asynchronous programming model

Planned integration between SCA and Spring will make Spring a natural way to build SCA components.

5.4 Java Enterprise Edition 5

The impending release of Java EE 5.0 in 2006 simplifies and improves ease of use characteristics in EJB 3.0. The most significant of the changes in EJB3 concerns dependency injection and the persistence model. The adoption of modern programming principles directly in Java EE 5.0 will result in good affinity between Java oriented component development with EJB 3.0 and service oriented assembly and composition with SCA, and is expected to provide EJB programmers with a more comprehensive platform for the implementation of solutions using an SOA.

5.4.1 EJB 3.0 Simplified Session Beans

EJB 3.0 simplified session beans can be used naturally as SCA component implementations. With EJB 3.0's dependency injection, it is possible for a reference to an SCA service to be injected onto an EJB 3.0 component, instead of requiring the EJB to use SCA client APIs to find the service reference. EJB 3.0 can also inject

environment entries onto the component. As in J2EE 1.4, these environment entries can be configured as SCA properties.

5.4.2 EJB 3.0 Persistence

EJB 3.0 introduces a new persistence model that is based on object-relational mapping technology. Unlike EJB2.1, EJB 3.0 entity beans are not remotable. Therefore, in order to use EJB 3.0 entity beans directly from an SCA component, the entity beans need to be in the same module as the SCA component. Also, the SCA container that is being used also needs to be a compliant EJB 3.0 container. When that is the case, the EJB 3.0 mechanisms for accessing entity managers, persistent contexts and entity beans can be used as described in the EJB 3.0 specification.

5.5 JMS and Messaging Systems

JMS may be used in two different ways within SCA:

1) As a transport for a service operation
2) As a provider of queue and topic services.

In both cases, JMS is handled via a Binding - <binding.jms…. />.

When JMS is used purely as the transport for a service operation, JMS queues are not visible artifacts within the development environment, although they are visible to administrators.

However, there are times when it makes sense for JMS queues to be exposed as explicit services that are available to the developer. For example, when several services draw work from the same work queue, the services that enqueue work are written to an API that is explicitly putting work on a queue. Similarly, a JMS topic must be represented as an explicit component, since message publishers publish to a topic.

The value of exposing JMS concepts as SCA concepts, rather than just accessing them through the JMS API, is that SCA can then be used for the static wiring between JMS clients and destinations. Thus there is the simplicity common model for integrating components through interfaces and wires that support both simple service invocations and also the pub/sub style commonly used with messaging systems. This also provides flexibility both when building components and when assembling systems since developers can provide components that can be integrated using synchronous call/return, asynchronous calls or pub/sub transparently to the component's implementation.

5.6 Web Services using JAX-RPC / JAX-WS and Axis

There are existing technologies for implementing Web services clients and implementations. In Java, there are the JAX-RPC and the new JAX-WS APIs, which can be used in conjunction with a variety of runtime technologies including J2EE servers and the popular Apache Axis runtime [6]. There is an Axis runtime also available for C++.

SCA components can be linked with Web services of these types using Web services bindings, both as clients and as implementations.

5.6.1 REST Web Services

REST [7] provides a simpler way of implementing basic Web services which can suit the needs of some SOA implementations. Over the last several years, some of the larger successful e-commerce portals, clearing houses and search engines such as Yahoo, Amazon and E-Bay have provided their customer facing service API's using plain old XML ("POX") or in a number of cases explicitly called their API's REST oriented. The goal of this has been to reduce the learning curve of their programming model and significantly simplify the overhead required to interact with their services.

SCA will support REST style Web services by supporting a REST Web services binding. SCA components can implement REST style Web services by means of offering their services using the REST binding. Similarly, SCA components can invoke REST style Web services through the REST binding. Other than a change of binding, using REST Web services is similar to the use of Web services using SOAP and WSDL bindings.

6 Future Directions

The current SCA specification is the foundation of a programming environment tuned to the needs of building SOA solutions. SCA also provides a model that unifies and simplifies many concepts, and enables a simplified suite of development tools. However, the current specification is published as a "0.9" level of specification and it is not intended to be a complete and final document. The current document concentrates on core capabilities and invites community feedback, especially on some of the more complex areas. Future versions of the specification will add capabilities in a number of significant areas and will reflect input from the wider community.

6.1 Bindings

The current bindings defined in the specification are limited to SCA binding and WSDL-described Web services. Additional bindings will be defined, including the following:

- JMS and Messaging bindings
- Queue and Topic bindings for messaging systems
- J2EE EJB bindings
- Web services bindings that conform to specific Profiles, such as the WS-I Basic Profile [4] and Basic Security Profile [5]

6.2 Implementation types

The current SCA spec defines implementation types for Java, local Java and for BPEL. This list is expected to grow quite rapidly to include implementations that use a variety of programming languages and technologies, including for example:

- C++
- PHP
- XSLT
- Java EJB
- XQuery
- SQL

6.3 Policy, Security, Transactions and Reliable Messaging

Important aspects of an SOA system in the areas of Security, Transactions and Reliable Messaging are not part of the main SCA specification at present. Much thought has gone into how to structure these often complex areas and this thinking is described in an Appendix to the specification – with the main emphasis on presenting a simple model to the end-programmer using Policies, while still preserving the flexibility required by the experts in these fields.

In future versions of the SCA specification, the areas covered in the Appendix will migrate into the main body of the specification, updated to reflect community feedback on the current proposals.

6.4 Asynchronous and Message-Oriented Model

The Asynchronous and Message-Oriented model for SCA services is currently not part of the main body of the SCA specification – it is an appendix which presents a detailed proposal for how these facilities will be presented under SCA. Asynchronous programming is a complex area and the writers of the SCA specification are looking for community feedback on these proposals before including this function in the main body of the specification.

7 References

[1] SCA Specification

Any one of:

- http://dev2dev.bea.com/technologies/commonj/index.jsp
- http://www.ibm.com/developerworks/library/specification/ws-sca/
- http://www.iona.com/devcenter/sca/
- http://oracle.com/technology/webservices/sca
- https://www.sdn.sap.com/
- http://www.sybase.com

[2] SDO Specification

- http://dev2dev.bea.com/technologies/commonj/index.jsp
- http://www.ibm.com/developerworks/library/specification/ws-sdo/
- http://oracle.com/technology/webservices/sca
- https://www.sdn.sap.com/
- http://www.xcalia/xdn/specs/sdo
- http:/www.sybase.com

[3] SCA Sample application “Building your first application – Simplified BigBank”

Any one of:

- http://dev2dev.bea.com/technologies/commonj/index.jsp
- http://www.ibm.com/developerworks/library/specification/ws-sca/
- http://www.iona.com/devcenter/sca/
- http://oracle.com/technology/webservices/sca
- https://www.sdn.sap.com/
- http://www.sybase.com

[4] WS-I Basic Profile
http://www.ws-i.org/deliverables/workinggroup.aspx?wg=basicprofile

[5] WS-I Basic Security Profile
http://www.ws-i.org/deliverables/workinggroup.aspx?wg=basicsecurity

[6] Apache Axis
http://ws.apache.org/axis/

[7] REST Web services
There is no single definitive source for REST Web services. The following articles represent good starting points and refer to multiple sources of information about REST-style Web services:
http://www.xfront.com/REST-Web-Services.html
and
http://www.xml.com/pub/a/2004/08/11/rest.html

[8] Enterprise Metadata Specification
http://www-128.ibm.com/developerworks/library/specification/j-emd/
or
http://dev2dev.bea.com/wlplatform/commonj/

[9] JAX-WS Specification
http://www.jcp.org/en/jsr/detail?id=224

[10] JAX-RPC Specification
http://jcp.org/en/jsr/detail?id=101

[11] Spring Framework
http://www.springframework.org/

[12] WSBPEL
http://www-128.ibm.com/developerworks/library/specification/ws-bpel/
http://www.oasis-open.org/committees/download.php/14616/wsbpel-specification-draft.htm

[13] Java Message Service (JMS)
http://java.sun.com/products/jms/docs.html

[14] J2EE Connector Architecture (JCA)
http://java.sun.com/j2ee/connector/download.html

[15] XSL Transformations (XSLT)
http://www.w3.org/TR/xslt

[16] XQuery Specification
http://www.w3.org/TR/2005/WD-xquery-20050915/

[17] WS-Policy Specification
http://www-128.ibm.com/developerworks/library/specification/ws-polfram/

[18] The Strategy Pattern
Design Patterns -- Elements of Reusable Object-Oriented Software
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides

[19] Web Services Definition Language (WSDL)
http://www.w3.org/TR/wsdl
http://www.w3.org/TR/wsdl20/