Service Component Architecture Assembly Model Specification Version 1.1

Working Draft

24 September, 2007

Specification URIs:
This Version:
http://docs.oasis-open.org/sca-assembly/sca-assembly-draft-20070924.html
http://docs.oasis-open.org/sca-assembly/sca-assembly-draft-20070924.doc
http://docs.oasis-open.org/sca-assembly/sca-assembly-draft-20070924.pdf

Previous Version:

Latest Version:
http://docs.oasis-open.org/sca-assembly/sca-assembly-draft-20070924.html
http://docs.oasis-open.org/sca-assembly/sca-assembly-draft-20070924.doc

Latest Approved Version:

Technical Committee:
OASIS Service Component Architecture / Assembly (SCA-Assembly) TC

Chair(s):
Martin Chapman, Oracle
Mike Edwards, IBM

Editor(s):
Michael Beisiegel, IBM
Khanderao Khand, Oracle
Anish Karmarkar, Oracle
Sanjay Patil, SAP
Michael Rowley, BEA Systems

Related work:

This specification replaces or supercedes:

- Service Component Architecture Assembly Model Specification Version 1.00, March 15, 2007

This specification is related to:

- Service Component Architecture Policy Framework Specification Version 1.1

Declared XML Namespace(s):

TBD

Abstract:

Service Component Architecture (SCA) provides a programming model for building applications and solutions based on a Service Oriented Architecture. It is based on the idea that business function is provided as a series of services, which are assembled together to create solutions that serve a particular business need. These composite applications can contain both new services created specifically for the application and also business function from existing systems and applications, reused as part of the composition. SCA provides a model both for the composition of services and for the creation of service components, including the reuse of existing application function within SCA composites.

SCA is a model that aims to encompass a wide range of technologies for service components and for the access methods which are used to connect them. For components, this includes not only different programming languages, but also frameworks and environments commonly used with those languages. For access methods, SCA compositions allow for the use of various communication and service access technologies that are in common use, including, for example, Web services, Messaging systems and Remote Procedure Call (RPC).

The SCA Assembly Model consists of a series of artifacts which define the configuration of an SCA domain in terms of composites which contain assemblies of service components and the connections and related artifacts which describe how they are linked together.

This document describes the SCA Assembly Model, which covers

- A model for the assembly of services, both tightly coupled and loosely coupled
- A model for applying infrastructure capabilities to services and to service interactions, including Security and Transactions

Status:

This document was last revised or approved by the OASIS Service Component Architecture / Assembly (SCA-Assembly) TC on the above date. The level of approval is also listed above. Check the “Latest Version” or “Latest Approved Version” location noted above for possible later revisions of this document.

Technical Committee members should send comments on this specification to the Technical Committee’s email list. Others should send comments to the Technical Committee by using the “Send A Comment” button on the Technical Committee’s web page at http://www.oasis-open.org/committees/sca-assembly/.

For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the Technical Committee web page (http://www.oasis-open.org/committees/sca-assembly/ipr.php.

The non-normative errata page for this specification is located at http://www.oasis-open.org/committees/sca-assembly/.

Notices

Table of Contents

1 Introduction

This document describes the ***SCA Assembly Model, which*** covers

- A model for the assembly of services, both tightly coupled and loosely coupled
- A model for applying infrastructure capabilities to services and to service interactions, including Security and Transactions

The document starts with a short overview of the SCA Assembly Model.

The next part of the document describes the core elements of SCA, SCA components and SCA composites.

The final part of the document defines how the SCA assembly model can be extended.

1.1 Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in **[RFC2119]**.

1.2 Normative References

[RFC2119]	S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, http://www.ietf.org/rfc/rfc2119.txt, IETF RFC 2119, March 1997.

[1] SCA Java Component Implementation Specification

SCA Java Common Annotations and APIs Specification

http://www.osoa.org/download/attachments/35/SCA_JavaComponentImplementation_V100.pdf

http://www.osoa.org/download/attachments/35/SCA_JavaAnnotationsAndAPIs_V100.pdf

[2] SDO Specification

http://www.osoa.org/download/attachments/36/Java-SDO-Spec-v2.1.0-FINAL.pdf

[3] SCA Example Code document

http://www.osoa.org/download/attachments/28/SCA_BuildingYourFirstApplication_V09.pdf

[4] JAX-WS Specification

http://jcp.org/en/jsr/detail?id=101

[5] WS-I Basic Profile

http://www.ws-i.org/deliverables/workinggroup.aspx?wg=basicprofile

[6] WS-I Basic Security Profile

http://www.ws-i.org/deliverables/workinggroup.aspx?wg=basicsecurity

[7] Business Process Execution Language (BPEL)

http://www.oasis-open.org/committees/documents.php?wg_abbrev=wsbpel

[8] WSDL Specification
WSDL 1.1: http://www.w3.org/TR/wsdl
WSDL 2.0: http://www.w3.org/TR/wsdl20/

[9] SCA Web Services Binding Specification
http://www.osoa.org/download/attachments/35/SCA_WebServiceBindings_V100.pdf

[10] SCA Policy Framework Specification
http://www.osoa.org/download/attachments/35/SCA_Policy_Framework_V100.pdf

[11] SCA JMS Binding Specification
http://www.osoa.org/download/attachments/35/SCA_JMSBinding_V100.pdf

[12] ZIP Format Definition
http://www.pkware.com/documents/casestudies/APPNOTE.TXT

[Reference] [Full reference citation]

2 Overview

Service Component Architecture (SCA) provides a programming model for building applications and solutions based on a Service Oriented Architecture. It is based on the idea that business function is provided as a series of services, which are assembled together to create solutions that serve a particular business need. These composite applications can contain both new services created specifically for the application and also business function from existing systems and applications, reused as part of the composition. SCA provides a model both for the composition of services and for the creation of service components, including the reuse of existing application function within SCA composites.

SCA is a model that aims to encompass a wide range of technologies for service components and for the access methods which are used to connect them. For components, this includes not only different programming languages, but also frameworks and environments commonly used with those languages. For access methods, SCA compositions allow for the use of various communication and service access technologies that are in common use, including, for example, Web services, Messaging systems and Remote Procedure Call (RPC).

The SCA ***Assembly Model*** consists of a series of artifacts which define the configuration of an SCA domain in terms of composites which contain assemblies of service components and the connections and related artifacts which describe how they are linked together.

One basic artifact of SCA is the ***component***, which is the unit of construction for SCA. A component consists of a configured instance of an implementation, where an implementation is the piece of program code providing business functions. The business function is offered for use by other components as ***services***. Implementations may depend on services provided by other components – these dependencies are called ***references***. Implementations can have settable ***properties***, which are data values which influence the operation of the business function. The component ***configures*** the implementation by providing values for the properties and by wiring the references to services provided by other components.

SCA allows for a wide variety of implementation technologies, including "traditional" programming languages such as Java, C++, and BPEL, but also scripting languages such as PHP and JavaScript and declarative languages such as XQuery and SQL.

SCA describes the content and linkage of an application in assemblies called ***composites***. Composites can contain components, services, references, property declarations, plus the wiring that describes the connections between these elements. Composites can group and link components built from different implementation technologies, allowing appropriate technologies to be used for each business task. In turn, composites can be used as complete component implementations: providing services, depending on references and with settable property values. Such composite implementations can be used in components within other composites, allowing for a hierarchical construction of business solutions, where high-level services are implemented internally by sets of lower-level services. The content of composites can also be used as groupings of elements which are contributed by inclusion into higher-level compositions.

Composites are deployed within an ***SCA Domain***. An SCA Domain typically represents a set of services providing an area of business functionality that is controlled by a single organization. As an example, for the accounts department in a business, the SCA Domain might cover all financial related function, and it might contain a series of composites dealing with specific areas of accounting, with one for customer accounts, another dealing with accounts payable. To help build and configure the SCA Domain, composites can be used to group and configure related artifacts.

SCA defines an XML file format for its artifacts. These XML files define the portable representation of the SCA artifacts. An SCA runtime may have other representations of the artifacts represented by these XML files. In particular, component implementations in some programming languages may have attributes or properties or annotations which can specify some of the elements of the SCA Assembly model. The XML files define a static format for the configuration of an SCA Domain. An SCA runtime may also allow for the configuration of the domain to be modified dynamically.

2.1 Diagram used to Represent SCA Artifacts

This document introduces diagrams to represent the various SCA artifacts, as a way of visualizing the relationships between the artifacts in a particular assembly. These diagrams are used in this document to accompany and illuminate the examples of SCA artifacts.

The following picture illustrates some of the features of an SCA component:

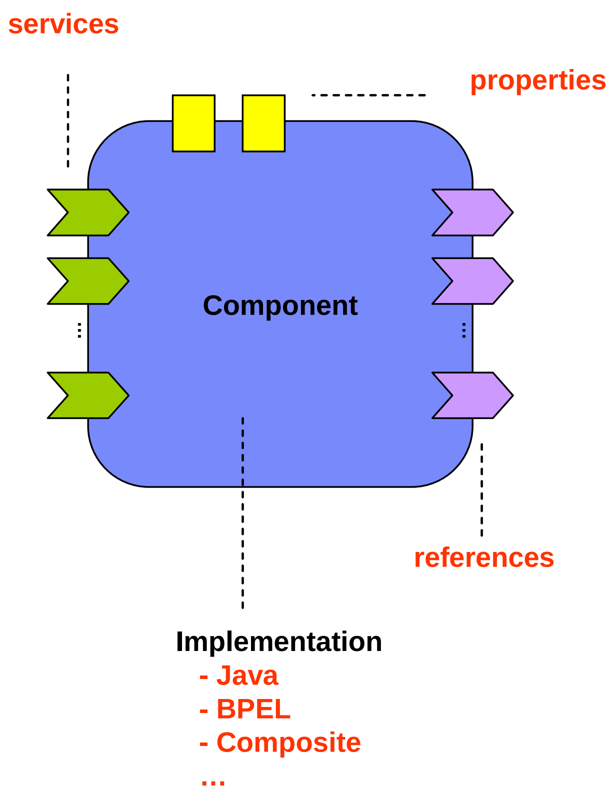

Figure 1: SCA Component Diagram

The following picture illustrates some of the features of a composite assembled using a set of components:

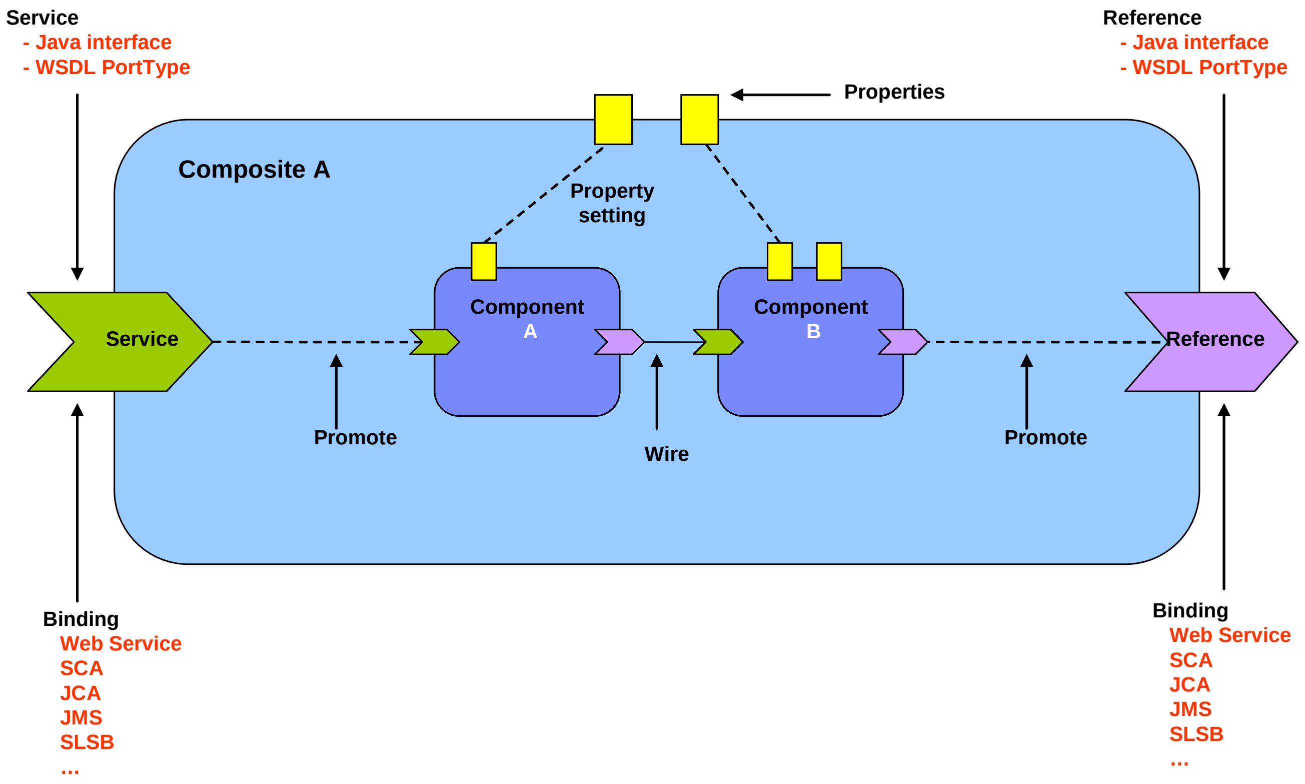

Figure 2: SCA Composite Diagram

The following picture illustrates an SCA Domain assembled from a series of high-level composites, some of which are in turn implemented by lower-level composites:

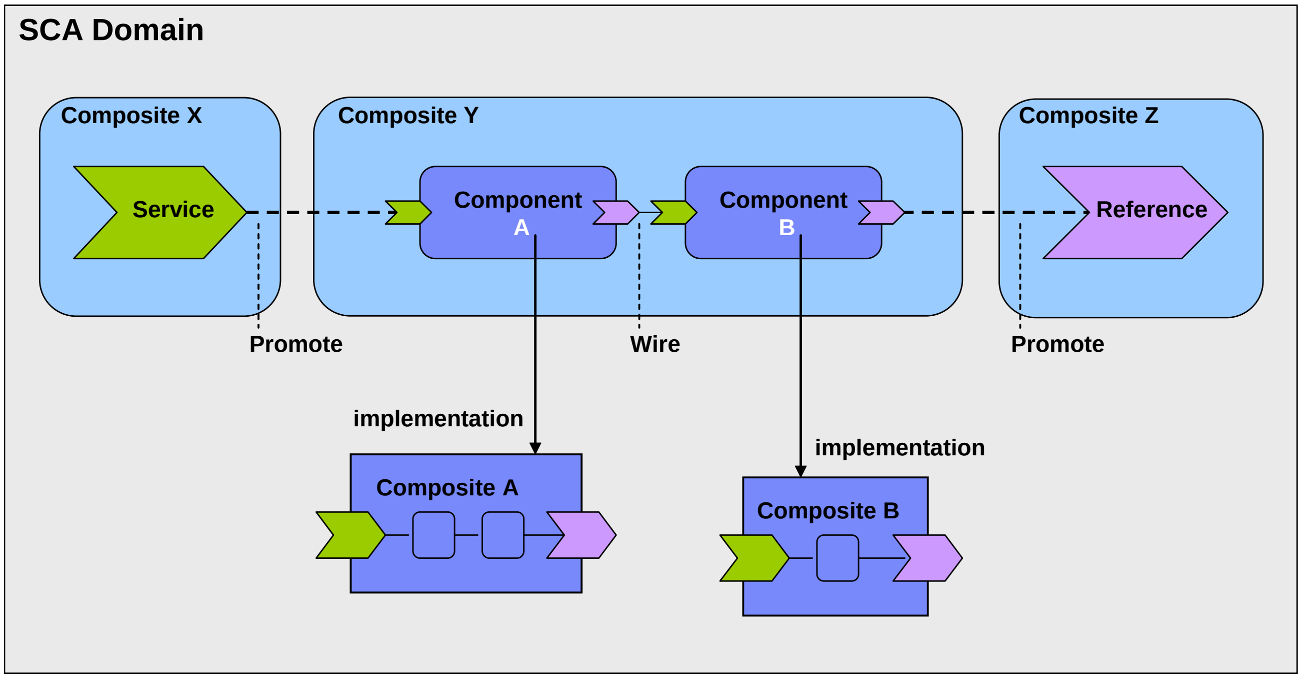

Figure 3: SCA Domain Diagram

3 Component

Components are the basic elements of business function in an SCA assembly, which are combined into complete business solutions by SCA composites.

Components are configured ***instances*** of ***implementations.*** Components provide and consume services. More than one component can use and configure the same implementation, where each component configures the implementation differently.

Components are declared as subelements of a composite in an ***xxx.composite*** file. A component is represented by a ***component element*** which is a child of the composite element. There can be ***zero or more*** component elements within a composite. The following snippet shows the composite schema with the schema for the component child element.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Component schema snippet -->
<composite     xmlns="http://www.osoa.org/xmlns/sca/1.0"
               targetNamespace="xs:anyURI"
               name="xs:NCName" local="xs:boolean"?
               autowire="xs:boolean"? constrainingType="QName"?
               requires="list of xs:QName"? policySets="list of
xs:QName"?>

   ...

   <component name="xs:NCName" requires="list of xs:QName"?
           autowire="xs:boolean"?
               requires="list of xs:QName"? policySets="list of xs:QName"?
               constrainingType="xs:QName"?>*
         <implementation/>?
         <service name="xs:NCName" requires="list of xs:QName"?
               policySets="list of xs:QName"?>*
               <interface/>?
               <binding uri="xs:anyURI"? requires="list of xs:QName"?
                     policySets="list of xs:QName"?/>*
         </service>
         <reference name="xs:NCName" multiplicity="0..1 or 1..1 or 0..n or
1..n"?
             autowire="xs:boolean"?
               target="list of xs:anyURI"? policySets="list of xs:QName"?
               wiredByImpl="xs:boolean"? requires="list of xs:QName"?>*
               <interface/>?
               <binding uri="xs:anyURI"? requires="list of xs:QName"?
                     policySets="list of xs:QName"?/>*
         </reference>
```

```
		<property name="xs:NCName" (type="xs:QName" |
element="xs:QName")?
			mustSupply="xs:boolean"?
			  many="xs:boolean"? source="xs:string"? file="xs:anyURI"?>*
			  property-value?
		</property>
	</component>

	...

</composite>
```

The component element has the following ***attributes***:

- ***name (required)*** – the name of the component. The name must be unique across all the components in the composite.
- ***autowire (optional)*** – whether contained component references should be autowired, as described in the Autowire section. Default is false.
- ***requires (optional)*** – a list of policy intents. See the Policy Framework specification [10] for a description of this attribute.
- ***policySets (optional)*** – a list of policy sets. See the Policy Framework specification [10] for a description of this attribute.
- ***constrainingType (optional)*** – the name of a constrainingType. When specified, the set of services, references and properties of the component, plus related intents, is constrained to the set defined by the constrainingType. See the ConstrainingType Section for more details.

A component element has ***zero or one implementation element*** as its child, which points to the implementation used by the component. A component with no implementation element is not runnable, but components of this kind may be useful during a "top-down" development process as a means of defining the characteristics required of the implementation before the implementation is written.

The component element can have ***zero or more service elements*** as children which are used to configure the services of the component. The services that can be configured are defined by the implementation.

The service element has the following ***attributes***:

- ***name (required)*** - the name of the service. Has to match a name of a service defined by the implementation.
- ***requires (optional)*** – a list of policy intents. See the Policy Framework specification [10] for a description of this attribute.
  Note: The effective set of policy intents for the service consists of any intents explicitly stated in this requires attribute, combined with any intents specified for the service by the implementation.
- ***policySets (optional)*** – a list of policy sets. See the Policy Framework specification [10] for a description of this attribute.

A service has ***zero or one interface***, which describes the operations provided by the service. The interface is described by an ***interface element*** which is a child element of the service element. If no interface is specified, then the interface specified for the service by the implementation is in effect. If an interface is specified it must provide a compatible subset of the interface provided by the implementation, i.e. provide a subset of the operations defined by the implementation for the service. For details on the interface element see the Interface section.

A service element has one or more ***binding elements*** as children. If no bindings are specified, then the bindings specified for the service by the implementation are in effect. If bindings are specified, then those bindings override the bindings specified by the implementation. Details of the binding element are described in the Bindings section. The binding, combined with any PolicySets in effect for the binding, must satisfy the set of policy intents for the service, as described in the Policy Framework specification [10].

The component element can have ***zero or more reference elements*** as children which are used to configure the references of the component. The references that can be configured are defined by the implementation.

The reference element has the following ***attributes***:

- ***name (required)*** – the name of the reference. Has to match a name of a reference defined by the implementation.
- ***autowire (optional)*** – whether the reference should be autowired, as described in the Autowire section. Default is false.
- ***requires (optional)*** – a list of policy intents. See the Policy Framework specification [10] for a description of this attribute.
  Note: The effective set of policy intents for the reference consists of any intents explicitly stated in this requires attribute, combined with any intents specified for the reference by the implementation.
- ***policySets (optional)*** – a list of policy sets. See the Policy Framework specification [10] for a description of this attribute.
- ***multiplicity (optional)*** - defines the number of wires that can connect the reference to target services. Overrides the multiplicity specified for this reference on the implementation. The value can only be equal or further restrict, i.e. 0..n to 0..1 or 1..n to 1..1. The multiplicity can have the following values
  - 1..1 – one wire can have the reference as a source
  - 0..1 – zero or one wire can have the reference as a source
  - 1..n – one or more wires can have the reference as a source
  - 0..n - zero or more wires can have the reference as a source
- ***target (optional)*** – a list of one or more of target service URI's, depending on multiplicity setting. Each value wires the reference to a component service that resolves the reference. For more details on wiring see the section on Wires. Overrides any target specified for this reference on the implementation.
- ***wiredByImpl (optional)*** – a boolean value, "false" by default, which indicates that the implementation wires this reference dynamically. If set to "true" it indicates that the target of the reference is set at runtime by the implementation code (eg by the code obtaining an endpoint reference by some means and setting this as the target of the reference through the use of programming interfaces defined by the relevant Client and Implementation specification). If "true" is set, then the reference should not be wired statically within a composite, but left unwired.

A reference has ***zero or one interface***, which describes the operations required by the reference. The interface is described by an ***interface element*** which is a child element of the reference element. If no interface is specified, then the interface specified for the reference by the implementation is in effect. If an interface is specified it must provide a compatible superset of the interface provided by the implementation, i.e. provide a superset of the operations defined by the implementation for the reference. For details on the interface element see the Interface section.

A reference element has one or more ***binding elements*** as children. If no bindings are specified, then the bindings specified for the reference by the implementation are in effect. If any bindings are specified, then those bindings override any and all the bindings specified by the implementation. Details of the binding element are described in the Bindings section. The binding, combined with any PolicySets in effect for the binding, must satisfy the set of policy intents for the reference, as described in the Policy Framework specification [10].

Note that a binding element may specify an endpoint which is the target of that binding. A reference must not mix the use of endpoints specified via binding elements with target endpoints specified via the target attribute. If the target attribute is set, then binding elements can only list one or more binding types that can be used for the wires identified by the target attribute. All the binding types identified are available for use on each wire in this case. If endpoints are specified in the binding elements, each endpoint must use the binding type of the binding element in which it is defined. In addition, each binding element needs to specify an endpoint in this case.

The component element has ***zero or more property elements*** as its children, which are used to configure data values of properties of the implementation. Each property element provides a value for the named property, which is passed to the implementation. The properties that can be configured and their types are defined by the implementation. An implementation can declare a property as multi-valued, in which case, multiple property values can be present for a given property.

The property value can be specified in ***one*** of three ways:

- As a value, supplied as the content of the property element
- By referencing a Property value of the composite which contains the component. The reference is made using the ***source*** attribute of the property element.

  The form of the value of the source attribute follows the form of an XPath expression. This form allows a specific property of the composite to be addressed by name. Where the property is complex, the XPath expression can be extended to refer to a sub-part of the complex value.

  So, for example, `source="$currency"` is used to reference a property of the composite called "currency", while `source="$currency/a"` references the sub-part "a" of the complex composite property with the name "currency".

- By specifying a dereferencable URI to a file containing the property value through the ***file*** attribute. The contents of the referenced file are used as the value of the property.

If more than one property value specification is present, the source attribute takes precedence, then the file attribute.

Optionally, the type of the property can be specified in ***one*** of two ways:

- by the qualified name of a type defined in an XML schema, using the **`type`** attribute
- by the qualified name of a global element in an XML schema, using the **`element`** attribute

The property type specified must be compatible with the type of the property declared by the implementation. If no type is specified, the type of the property declared by the implementation is used.

The property element has the following attributes:

- ***name (required)*** – the name of the property. Has to match a name of a property defined by the implementation
- ***type (optional)*** – the type of the property defined as the qualified name of an XML schema type
- ***element (optional)*** – the type of the property defined as the qualified name of an XML schema global element – the type is the type of the global element
- ***source (optional)*** – an XPath expression pointing to a property of the containing composite from which the value of this component property is obtained.
- ***file (optional)*** – a dereferencable URI to a file containing a value for the property
- ***many (optional)*** – (optional) whether the property is single-valued (false) or multi-valued (true). Overrides the many specified for this property on the implementation. The value can only be equal or further restrict, i.e. if the implementation specifies many true, then the component can say false. In the case of a multi-valued property, it is presented to the implementation as a Collection of property values.

3.1 Example Component

The following figure shows the ***component symbol*** that is used to represent a component in an assembly diagram.

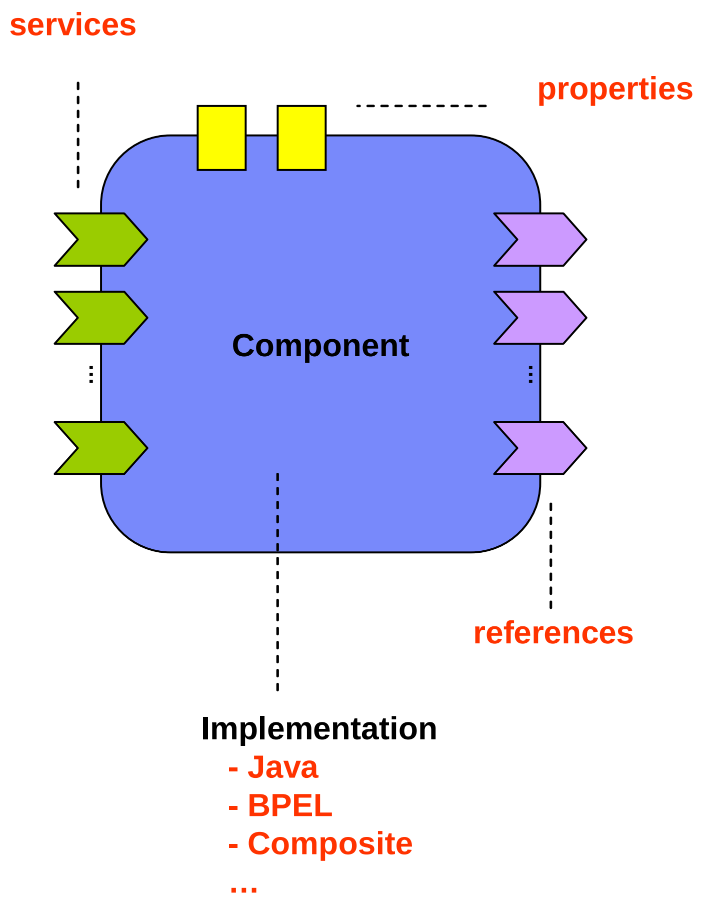

Figure 4: Component symbol

The following figure shows the assembly diagram for the MyValueComposite containing the MyValueServiceComponent.

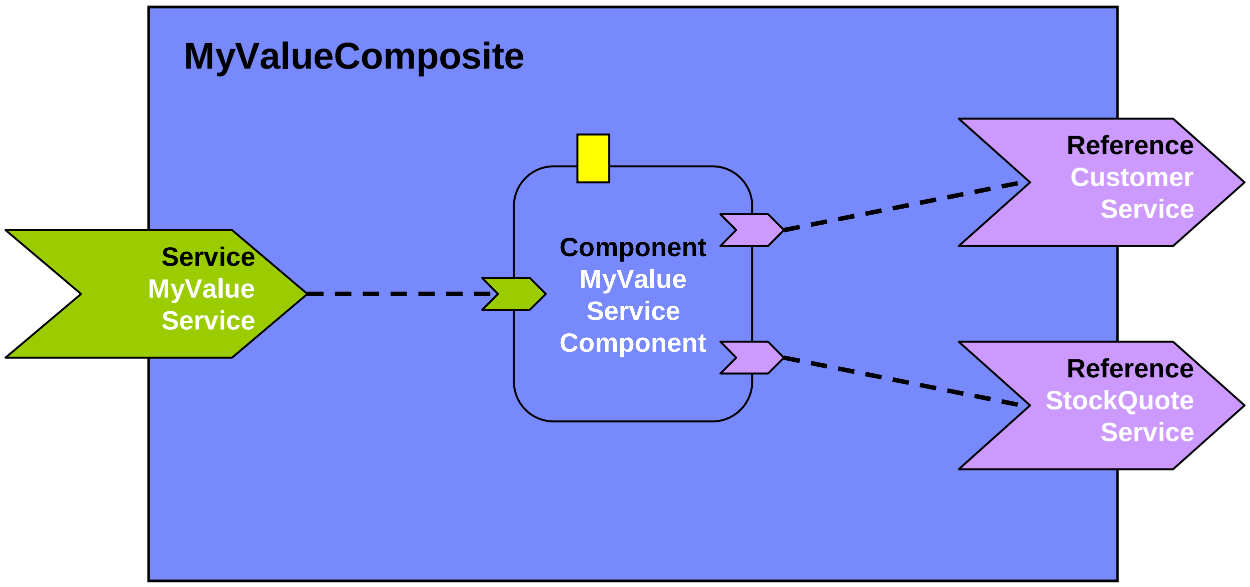

Figure 5: Assembly diagram for MyValueComposite

The following snippet shows the MyValueComposite.composite file for the MyValueComposite containing the component element for the MyValueServiceComponent. A value is set for the property named currency, and the customerService and stockQuoteService references are promoted:

```
<?xml version="1.0" encoding="ASCII"?>
<!-- MyValueComposite_1 example -->
<composite     xmlns="http://www.osoa.org/xmlns/sca/1.0"
               targetNamespace="http://foo.com"
               name="MyValueComposite" >

   <service name="MyValueService" promote="MyValueServiceComponent"/>

   <component name="MyValueServiceComponent">
         <implementation.java
class="services.myvalue.MyValueServiceImpl"/>
         <property name="currency">EURO</property>
         <reference name="customerService"/>
         <reference name="stockQuoteService"/>
   </component>

   <reference name="CustomerService"
         promote="MyValueServiceComponent/customerService"/>
   <reference name="StockQuoteService"
         promote="MyValueServiceComponent/stockQuoteService"/>

</composite>
```

Note that the references of MyValueServiceComponent are explicitly declared only for purposes of clarity – the references are defined by the MyValueServiceImpl implementation and there is no need to redeclare them on the component unless the intention is to wire them or to override some aspect of them.

The following snippet gives an example of the layout of a composite file if both the currency property and the customerService reference of the MyValueServiceComponent are declared to be multi-valued (many=true for the property and multiplicity=0..n or 1..n for the reference):

```
<?xml version="1.0" encoding="ASCII"?>
<!-- MyValueComposite_2 example -->
<composite     xmlns="http://www.osoa.org/xmlns/sca/1.0"
               targetNamespace="http://foo.com"
               name="MyValueComposite" >

   <service name="MyValueService" promote="MyValueServiceComponent"/>

   <component name="MyValueServiceComponent">
         <implementation.java
class="services.myvalue.MyValueServiceImpl"/>
         <property name="currency">EURO</property>
         <property name="currency">Yen</property>
         <property name="currency">USDollar</property>
         <reference name="customerService"
               target="InternalCustomer/customerService"/>
         <reference name="StockQuoteService"/>
   </component>

   ...

   <reference name="CustomerService"
         promote="MyValueServiceComponent/customerService"/>

   <reference name="StockQuoteService"
         promote="MyValueServiceComponent/StockQuoteService"/>

</composite>
```

….this assumes that the composite has another component called InternalCustomer (not shown) which has a service to which the customerService reference of the MyValueServiceComponent is wired as well as being promoted externally through the composite reference CustomerService.

4 Implementation

Component ***implementations*** are concrete implementations of business function which provide services and/or which make references to services provided elsewhere. In addition, an implementation may have some settable property values.

SCA allows you to choose from any one of a wide range of ***implementation types***, such as Java, BPEL or C++, where each type represents a specific implementation technology. The technology may not simply define the implementation language, such as Java, but may also define the use of a specific framework or runtime environment. Examples include Java implementations done using the Spring framework or the Java EE EJB technology.

For example, within a component declaration in a composite file, the elements ***implementation.java*** and ***implementation.bpel*** point to Java and BPEL implementation types respectively. ***implementation.composite*** points to the use of an SCA composite as an implementation. ***implementation.spring*** and ***implementation.ejb*** are used for Java components written to the Spring framework and the Java EE EJB technology respectively.

The following snippets show implementation elements for the Java and BPEL implementation types and for the use of a composite as an implementation:

```
<implementation.java class="services.myvalue.MyValueServiceImpl"/>

<implementation.bpel process="MoneyTransferProcess"/>

<implementation.composite name="MyValueComposite"/>
```

Services, references and properties are the configurable aspects of an implementation. SCA refers to them collectively as the ***component type***. The characteristics of services, references and properties are described in the Component section. Depending on the implementation type, the implementation may be able to declare the services, references and properties that it has and it also may be able to set values for all the characteristics of those services, references and properties.

So, for example:

- for a service, the implementation may define the interface, binding(s), a URI, intents, and policy sets, including details of the bindings
- for a reference, the implementation may define the interface, binding(s), target URI(s), intents, policy sets, including details of the bindings
- for a property the implementation may define its type and a default value
- the implementation itself may define intents and policy sets

Most of the characteristics of the services, references and properties may be overridden by a component that uses and configures the implementation, or the component can decide not to override those characteristics. Some characteristics cannot be overridden, such as intents. Other characteristics, such as interfaces, can only be overridden in particular controlled ways (see the Component section for details).

The means by which an implementation declares its services, references and properties depend on the type of the implementation. For example, some languages, like Java, provide annotations which can be used to declare this information inline in the code.

At runtime, an ***implementation instance*** is a specific runtime instantiation of the implementation – its runtime form depends on the implementation technology used. The implementation instance derives its business logic from the implementation on which it is based, but the values for its properties and references are derived from the component which configures the implementation.

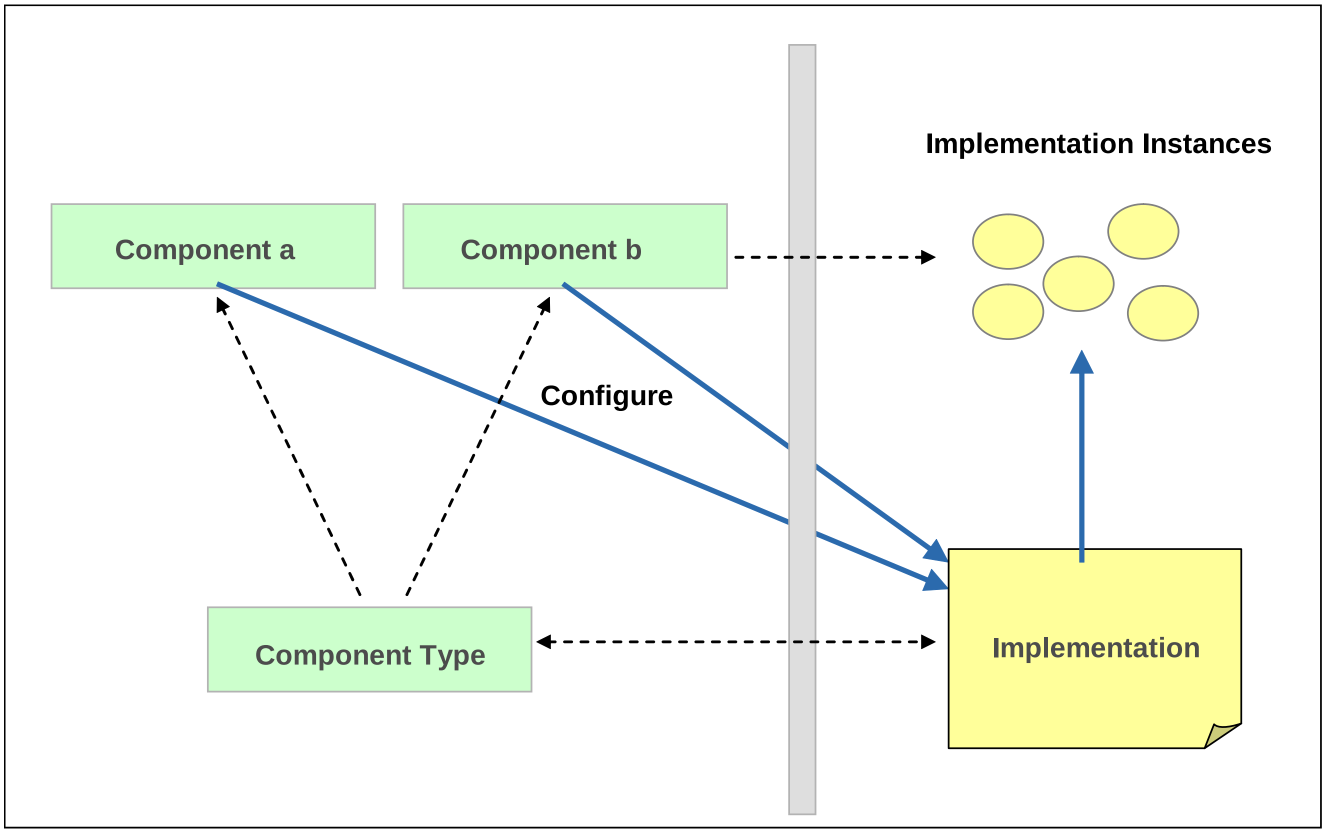

Figure 6: Relationship of Component and Implementation

4.1 Component Type

Component type represents the configurable aspects of an implementation. A component type consists of services that are offered, references to other services that can be wired and properties that can be set. The settable properties and the settable references to services are configured by a component which uses the implementation.

The ***component type is calculated in two steps*** where the second step adds to the information found in the first step. Step one is introspecting the implementation (if possible), including the inspection of implementation annotations (if available). Step two covers the cases where introspection of the implementation is not possible or where it does not provide complete information and it involves looking for an SCA ***component type file***. Component type information found in the component type file must be compatible with the equivalent information found from inspection of the implementation. The component type file can specify partial information, with the remainder being derived from the implementation.

In the ideal case, the component type information is determined by inspecting the implementation, for example as code annotations. The component type file provides a mechanism for the provision of component type information for implementation types where the information cannot be determined by inspecting the implementation.

A ***component type file*** has the same name as the implementation file but has the extension "***.componentType***". The component type is defined by a ***componentType element*** in the file. The ***location*** of the component type file depends on the type of the component implementation: it is described in the respective client and implementation model specification for the implementation type.

The componentType element can contain Service elements, Reference elements and Property elements.

The following snippet shows the componentType schema.

```
<?xml version="1.0" encoding="ASCII"?>
<!-- Component type schema snippet -->
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0"
   constrainingType="QName"? >

   <service name="xs:NCName" requires="list of xs:QName"?
         policySets="list of xs:QName"?>*
         <interface/>
         <binding uri="xs:anyURI"? requires="list of xs:QName"?
               policySets="list of xs:QName"?/>*
   </service>

   <reference name="xs:NCName" target="list of xs:anyURI"?
         multiplicity="0..1 or 1..1 or 0..n or 1..n"?
         wiredByImpl="xs:boolean"? requires="list of xs:QName"?
         policySets="list of xs:QName"?>*
         <interface/>?
         <binding uri="xs:anyURI"? requires="list of xs:QName"?
               policySets="list of xs:QName"?/>*
   </reference>

   <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
           many="xs:boolean"? mustSupply="xs:boolean"?
           policySets="list of xs:QName"?>*
         default-property-value?
   </property>

   <implementation requires="list of xs:QName"?
         policySets="list of xs:QName"?/>?

</componentType>
```

ComponentType has a single attribute:

- ***constrainingType (optional)*** – the name of a constrainingType. When specified, the set of services, references and properties of the implementation, plus related intents, is constrained to the set defined by the constrainingType. See the ConstrainingType Section for more details.

A Service represents an addressable interface of the implementation. The service is represented by a ***service element*** which is a child of the componentType element. There can be ***zero or more*** service elements in a componentType. See the Service section for details.

A ***Reference*** represents a requirement that the implementation has on a service provided by another component. The reference is represented by a ***reference element*** which is a child of the componentType element. There can be ***zero or more*** reference elements in a component type definition. See the Reference section for details.

Properties allow for the configuration of an implementation with externally set values. Each Property is defined as a property element. The componentType element can have zero or more property elements as its children. See the Property section for details.

Implementation represents characteristics inherent to the implementation itself, in particular intents and policies. See the Policy Framework specification [10] for a description of intents and policies.

4.1.1 Example ComponentType

The following snippet shows the contents of the componentType file for the MyValueServiceImpl implementation. The componentType file shows the services, references, and properties of the MyValueServiceImpl implementation. In this case, Java is used to define interfaces:

```
<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0">

   <service name="MyValueService">
         <interface.java interface="services.myvalue.MyValueService"/>
   </service>

   <reference name="customerService">
         <interface.java interface="services.customer.CustomerService"/>
   </reference>
   <reference name="stockQuoteService">
         <interface.java
interface="services.stockquote.StockQuoteService"/>
   </reference>

   <property name="currency" type="xsd:string">USD</property>

</componentType>
```

4.1.2 Example Implementation

The following is an example implementation, written in Java. See the SCA Example Code document [3] for details.

AccountServiceImpl implements the ***AccountService*** interface, which is defined via a Java interface:

```
package services.account;

@Remotable
public interface AccountService{

    public AccountReport getAccountReport(String customerID);
}
```

The following is a full listing of the AccountServiceImpl class, showing the Service it implements, plus the service references it makes and the settable properties that it has. Notice the use of Java annotations to mark SCA aspects of the code, including the @Property and @Reference tags:

```
package services.account;

import java.util.List;

import commonj.sdo.DataFactory;

import org.osoa.sca.annotations.Property;
import org.osoa.sca.annotations.Reference;

import services.accountdata.AccountDataService;
import services.accountdata.CheckingAccount;
import services.accountdata.SavingsAccount;
import services.accountdata.StockAccount;
import services.stockquote.StockQuoteService;

public class AccountServiceImpl implements AccountService {

    @Property
    private String currency = "USD";

    @Reference
    private AccountDataService accountDataService;
    @Reference
    private StockQuoteService stockQuoteService;

    public AccountReport getAccountReport(String customerID) {

     DataFactory dataFactory = DataFactory.INSTANCE;
     AccountReport accountReport = (AccountReport)dataFactory.create(AccountReport.class);
     List accountSummaries = accountReport.getAccountSummaries();

     CheckingAccount checkingAccount = accountDataService.getCheckingAccount(customerID);
     AccountSummary checkingAccountSummary =
(AccountSummary)dataFactory.create(AccountSummary.class);
     checkingAccountSummary.setAccountNumber(checkingAccount.getAccountNumber());
     checkingAccountSummary.setAccountType("checking");
     checkingAccountSummary.setBalance(fromUSDollarToCurrency(checkingAccount.getBalance()));
```

```
        accountSummaries.add(checkingAccountSummary);

        SavingsAccount savingsAccount = accountDataService.getSavingsAccount(customerID);
        AccountSummary savingsAccountSummary =
(AccountSummary)dataFactory.create(AccountSummary.class);
        savingsAccountSummary.setAccountNumber(savingsAccount.getAccountNumber());
        savingsAccountSummary.setAccountType("savings");
        savingsAccountSummary.setBalance(fromUSDollarToCurrency(savingsAccount.getBalance()));
        accountSummaries.add(savingsAccountSummary);

        StockAccount stockAccount = accountDataService.getStockAccount(customerID);
        AccountSummary stockAccountSummary =
(AccountSummary)dataFactory.create(AccountSummary.class);
        stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());
        stockAccountSummary.setAccountType("stock");
        float balance=
(stockQuoteService.getQuote(stockAccount.getSymbol()))*stockAccount.getQuantity();
        stockAccountSummary.setBalance(fromUSDollarToCurrency(balance));
        accountSummaries.add(stockAccountSummary);

        return accountReport;
       }

       private float fromUSDollarToCurrency(float value){

        if (currency.equals("USD")) return value; else
        if (currency.equals("EURO")) return value * 0.8f; else
        return 0.0f;
       }
    }
```

The following is the equivalent SCA componentType definition for the AccountServiceImpl, derived by reflection aginst the code above:

```
<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <service name="AccountService">
         <interface.java interface="services.account.AccountService"/>
   </service>
   <reference name="accountDataService">
         <interface.java
interface="services.accountdata.AccountDataService"/>
   </reference>
   <reference name="stockQuoteService">
         <interface.java
interface="services.stockquote.StockQuoteService"/>
   </reference>
```

```
    <property name="currency" type="xsd:string">USD</property>

</componentType>
```

For full details about Java implementations, see the Java Client and Implementation Specification and the SCA Example Code document. Other implementation types have their own specification documents.

5 Interface

Interfaces define one or more business functions. These business functions are provided by Services and are used by References. A Service offers the business functionality of exactly one interface for use by other components. Each interface defines one or more service ***operations*** and each operation has zero or one ***request (input) message*** and zero or one ***response (output) message***. The request and response messages may be simple types such as a string value or they may be complex types.

SCA currently supports the following interface type systems:

- Java interfaces
- WSDL 1.1 portTypes
- WSDL 2.0 interfaces

(WSDL: Web Services Definition Language [8])

SCA is also extensible in terms of interface types. Support for other interface type systems can be added through the extensibility mechanisms of SCA, as described in the Extension Model section.

The following snippet shows the schema for the Java interface element.

```
<interface.java interface="NCName" … />
```

The interface.java element has the following attributes:

- ***interface*** – the fully qualified name of the Java interface

The following sample shows a sample for the Java interface element.

```
<interface.java interface="services.stockquote.StockQuoteService"/>
```

Here, the Java interface is defined in the Java class file ./services/stockquote/StockQuoteService.class, where the root directory is defined by the contribution in which the interface exists.

For the Java interface type system, ***arguments and return*** of the service methods are described using Java classes or simple Java types. Service Data Objects [2] are the preferred form of Java class because of their integration with XML technologies.

For more information about Java interfaces, including details of SCA-specific annotations, see the Java Client and Implementation specification [1].

The following snippet shows a sample for the WSDL portType (WSDL 1.1) or WSDL interface (WSDL 2.0) element.

```
<interface.wsdl interface="xs:anyURI" … />
```

The interface.wsdl element has the following attributes:

- ***interface*** – URI of the portType/interface with the following format
    - <WSDL-namespace-URI>#wsdl.interface(<portTypeOrInterface-name>)

The following snippet shows a sample for the WSDL portType/interface element.

```
<interface.wsdl interface=”http://www.stockquote.org/StockQuoteService#
                                                          wsdl.interface(StockQuo
                                                    te)”/>
```

For WSDL 1.1, the interface attribute points to a portType in the WSDL. For WSDL 2.0, the interface attribute points to an interface in the WSDL. For the WSDL 1.1 portType and WSDL 2.0 interface type systems, arguments and return of the service operations are described using XML schema.

5.1 Local and Remotable Interfaces

A remotable service is one which may be called by a client which is running in an operating system process different from that of the service itself (this also applies to clients running on different machines from the service). Whether a service of a component implementation is remotable is defined by the interface of the service. In the case of Java this is defined by adding the ***@Remotable*** annotation to the Java interface (see Client and Implementation Model Specification for Java). WSDL defined interfaces are always remotable.

The style of remotable interfaces is typically ***coarse grained*** and intended for ***loosely coupled*** interactions. Remotable service Interfaces MUST NOT make use of ***method or operation overloading***.

Independent of whether the remotable service is called remotely from outside the process where the service runs or from another component running in the same process, the data exchange semantics are ***by-value***.

Implementations of remotable services may modify input messages (parameters) during or after an invocation and may modify return messages (results) after the invocation. If a remotable service is called locally or remotely, the SCA container is responsible for making sure that no modification of input messages or post-invocation modifications to return messages are seen by the caller.

Here is a snippet which shows an example of a remotable java interface:

```
package services.hello;

@Remotable
public interface HelloService {

   String hello(String message);
}
```

It is possible for the implementation of a remotable service to indicate that it can be called using by-reference data exchange semantics when it is called from a component in the same process. This can be used to improve performance for service invocations between components that run in the same process. This can be done using the @AllowsPassByReference annotation (see the Java Client and Implementation Specification).

A service typed by a local interface can only be called by clients that are running in the same process as the component that implements the local service. Local services cannot be published via remotable services of a containing composite. In the case of Java a local service is defined by a Java interface definition without a ***@Remotable*** annotation.

The style of local interfaces is typically ***fine grained*** and intended for ***tightly coupled*** interactions. Local service interfaces can make use of ***method or operation overloading***.

The data exchange semantic for calls to services typed by local interfaces is ***by-reference***.

5.2 Bidirectional Interfaces

The relationship of a business service to another business service is often peer-to-peer, requiring a two-way dependency at the service level. In other words, a business service represents both a consumer of a service provided by a partner business service and a provider of a service to the partner business service. This is especially the case when the interactions are based on asynchronous messaging rather than on remote procedure calls. The notion of ***bidirectional interfaces*** is used in SCA to directly model peer-to-peer bidirectional business service relationships.

An interface element for a particular interface type system must allow the specification of an optional callback interface. If a callback interface is specified SCA refers to the interface as a whole as a bidirectional interface.

The following snippet shows the interface element defined using Java interfaces with an optional callbackInterface attribute.

```
<interface.java      interface="services.invoicing.ComputePrice"
                     callbackInterface="services.invoicing.InvoiceCallback"/>
```

If a service is defined using a bidirectional interface element then its implementation implements the interface, and its implementation uses the callback interface to converse with the client that called the service interface.

If a reference is defined using a bidirectional interface element, the client component implementation using the reference calls the referenced service using the interface. The client component implementation must implement the callback interface.

Callbacks may be used for both remotable and local services. Either both interfaces of a bidirectional service MUST be remotable, or both MUST be local. A bidirectional service MUST NOT mix local and remote services.

Facilities are provided within SCA which allow a different component to provide a callback interface than the component which was the client to an original service invocation. How this is done can be seen in the SCA Java Client and Implementation Specification (section named “Passing Conversational Services as Parameters”).

5.3 Conversational Interfaces

Services sometimes cannot easily be defined so that each operation stands alone and is completely independent of the other operations of the same service. Instead, there is a sequence of operations that must be called in order to achieve some higher level goal. SCA calls this sequence of operations a ***conversation***. If the service uses a bidirectional interface, the conversation may include both operations and callbacks.

Such conversational services are typically managed by using conversation identifiers that are either (1) part of the application data (message parts or operation parameters) or 2) communicated separately from application data (possibly in headers). SCA introduces the concept of *conversational interfaces* for describing the interface contract for conversational services of the second form above. With this form, it is possible for the runtime to automatically manage the conversation, with the help of an appropriate binding specified at deployment. SCA does not standardize any aspect of conversational services that are maintained using application data. Such services are neither helped nor hindered by SCA's conversational service support.

Conversational services typically involve state data that relates to the conversation that is taking place. The creation and management of the state data for a conversation has a significant impact on the development of both clients and implementations of conversational services.

Traditionally, application developers who have needed to write conversational services have been required to write a lot of plumbing code. They need to:

- choose or define a protocol to communicate conversational (correlation) information between the client & provider
- route conversational messages in the provider to a machine that can handle that conversation, while handling concurrent data access issues
- write code in the client to use/encode the conversational information
- maintain state that is specific to the conversation, sometimes persistently and transactionally, both in the implementation and the client.

SCA makes it possible to divide the effort associated with conversational services between a number of roles:

- Application Developer: Declares that a service interface is conversational (leaving the details of the protocol up to the binding). Uses lifecycle semantics, APIs or other programmatic mechanisms (as defined by the implementation-type being used) to manage conversational state.
- Application Assembler: chooses a binding that can support conversations
- Binding Provider: implements a protocol that can pass conversational information with each operation request/response.
- Implementation-Type Provider: defines APIs and/or other programmatic mechanisms for application developers to access conversational information. Optionally implements instance lifecycle semantics that automatically manage implementation state based on the binding's conversational information.

This specification requires interfaces to be marked as conversational by means of a policy intent with the name ***"conversational"***. The form of the marking of this intent depends on the interface type. Note that it is also possible for a service or a reference to set the conversational intent when using an interface which is not marked with the conversational intent. This can be useful when reusing an existing interface definition that does not contain SCA information.

The meaning of the conversational intent is that both the client and the provider of the interface may assume that messages (in either direction) will be handled as part of an ongoing conversation without depending on identifying information in the body of the message (i.e. in parameters of the operations). In effect, the conversation interface specifies a high-level abstract protocol that must be satisfied by any actual binding/policy combination used by the service.

Examples of binding/policy combinations that support conversational interfaces are:

- Web service binding with a WS-RM policy
- Web service binding with a WS-Addressing policy
- Web service binding with a WS-Context policy
- JMS binding with a conversation policy that uses the JMS correlationID header

Conversations occur between one client and one target service. Consequently, requests originating from one client to multiple target conversational services will result in multiple conversations. For example, if a client A calls services B and C, both of which implement conversational interfaces, two conversations result, one between A and B and another between A and C. Likewise, requests flowing through multiple implementation instances will result in multiple conversations. For example, a request flowing from A to B and then from B to C will involve two conversations (A and B, B and C). In the previous example, if a request was then made from C to A, a third conversation would result (and the implementation instance for A would be different from the one making the original request).

Invocation of any operation of a conversational interface MAY start a conversation. The decision on whether an operation would start a conversation depends on the component's implementation and its implementation type. Implementation types MAY support components with conversational services. If an implementation type does provide this support, it must provide a mechanism for determining when a new conversation should be used for an operation (for example, in Java, the conversation is new on the first use of an injected reference; in BPEL, the conversation is new when the client's partnerLink comes into scope).

One or more operations in a conversational interface may be annotated with an *endsConversation* annotation (the mechanism for annotating the interface depends on the interface type). Where an interface is ***bidirectional***, operations may also be annotated in this way on operations of a callback interface. When a conversation ending operation is called, it indicates to both the client and the service provider that the conversation is complete. Any subsequent attempts to call an operation or a callback operation associated with the same conversation will generate a sca:ConversationViolation fault.

A sca:ConversationViolation fault is thrown when one of the following errors occurr:

- A message is received for a particular conversation, after the conversation has ended
- The conversation identification is invalid (not unique, out of range, etc.)
- The conversation identification is not present in the input message of the operation that ends the conversation
- The client or the service attempts to send a message in a conversation, after the conversation has ended

This fault is named within the SCA namespace standard prefix "sca", which corresponds to URI `http://www.osoa.org/xmlns/sca/1.0.`

The lifecycle of resources and the association between unique identifiers and conversations are determined by the service's implementation type and may not be directly affected by the "endConversation" annotation. For example, a WS-BPEL process may outlive most of the conversations that it is involved in.

Although conversational interfaces do not require that any identifying information be passed as part of the body of messages, there is conceptually an identity associated with the conversation. Individual implementations types MAY provide an API to access the ID associated with the conversation, although no assumptions may be made about the structure of that identifier. Implementation types MAY also provide a means to set the conversation ID by either the client or the service provider, although the operation may only be supported by some binding/policy combinations.

Implementation-type specifications are encouraged to define and provide conversational instance lifecycle management for components that implement conversational interfaces. However, implementations may also manage the conversational state manually.

5.4 SCA-Specific Aspects for WSDL Interfaces

There are a number of aspects that SCA applies to interfaces in general, such as marking them ***conversational***. These aspects apply to the interfaces themselves, rather than their use in a specific place within SCA. There is thus a need to provide appropriate ways of marking the interface definitions themselves, which go beyond the basic facilities provided by the interface definition language.

For WSDL interfaces, there is an extension mechanism that permits additional information to be included within the WSDL document. SCA takes advantage of this extension mechanism. In order to use the SCA extension mechanism, the SCA namespace (http://www.osoa.org/xmlns/sca/1.0) must be declared within the WSDL document.

First, SCA defines a global attribute in the SCA namespace which provides a mechanism to attach policy intents - ***@requires***. The definition of this attribute is as follows:

```
 <attribute name="requires" type="sca:listOfQNames"/>

 <simpleType name="listOfQNames">
   <list itemType="QName"/>
 </simpleType>
```

The @requires attribute can be applied to WSDL Port Type elements (WSDL 1.1) and to WSDL Interface elements (WSDL 2.0). The attribute contains one or more intent names, as defined by the Policy Framework specification [10]. Any service or reference that uses an interface with required intents implicitly adds those intents to its own @requires list.

To specify that a WSDL interface is conversational, the following attribute setting is used on either the WSDL Port Type or WSDL Interface:

```
requires="conversational"
```

SCA defines an ***endsConversation*** attribute that is used to mark specific operations within a WSDL interface declaration as ending a conversation. This only has meaning for WSDL interfaces which are also marked conversational. The endsConversation attribute is a global attribute in the SCA namespace, with the following definition:

```
   <attribute name="endsConversation" type="boolean" default="false"/>
```

The following snippet is an example of a WSDL Port Type annotated with the ***requires*** attribute on the portType and the ***endsConversation*** attribute on one of the operations:

```
   ...
   <portType name="LoanService" sca:requires="conversational">
       <operation name="apply">
          <input message="tns:ApplicationInput"/>
          <output message="tns:ApplicationOutput"/>
       </operation>
       <operation name="cancel" sca:endsConversation="true">
       </operation>
```

```
        ...
    </portType>
    ...
```

6 Composite

An SCA composite is used to assemble SCA elements in logical groupings. It is the basic unit of composition within an SCA Domain. An ***SCA composite*** contains a set of components, services, references and the wires that interconnect them, plus a set of properties which can be used to configure components.

Composites may form ***component implementations*** in higher-level composites – in other words the higher-level composites can have components that are implemented by composites. For more detail on the use of composites as component implementations see the section Using Composites as Component Implementations.

The content of a composite may be used within another composite through ***inclusion***. When a composite is included by another composite, all of its contents are made available for use within the including composite – the contents are fully visible and can be referenced by other elements within the including composite. For more detail on the inclusion of one composite into another see the section Using Composites through Inclusion.

A composite can be used as a unit of deployment. When used in this way, composites contribute elements to an SCA domain. A composite can be deployed to the SCA domain either by inclusion, or a composite can be deployed to the domain as an implementation. For more detail on the deployment of composites, see the section dealing with the SCA Domain.

A composite is defined in an ***xxx.composite*** file. A composite is represented by a ***composite*** element. The following snippet shows the schema for the composite element.

```
<?xml version="1.0" encoding="ASCII"?>
<!-- Composite schema snippet -->
<composite     xmlns="http://www.osoa.org/xmlns/sca/1.0"
               targetNamespace="xs:anyURI"
               name="xs:NCName" local="xs:boolean"?
               autowire="xs:boolean"? constrainingType="QName"?
               requires="list of xs:QName"? policySets="list of
xs:QName"?>

   <include name="xs:QName"/>*

   <service name="xs:NCName" promote="xs:anyURI"
         requires="list of xs:QName"? policySets="list of xs:QName"?>*
         <interface/>?
         <binding uri="xs:anyURI"? name="xs:QName"?
               requires="list of xs:QName"? policySets="list of
xs:QName"?/>*
         <callback>?
               <binding uri="xs:anyURI"? name="xs:QName"?
                     requires="list of xs:QName"?
                     policySets="list of xs:QName"?/>+
        </callback>
   </service>

   <reference name="xs:NCName" target="list of xs:anyURI"?
         promote="list of xs:anyURI" wiredByImpl="xs:boolean"?
         multiplicity="0..1 or 1..1 or 0..n or 1..n"?
         requires="list of xs:QName"? policySets="list of xs:QName"?>*
         <interface/>?
         <binding uri="xs:anyURI"? name="xs:QName"?
               requires="list of xs:QName"? policySets="list of
xs:QName"?/>*
         <callback>?
               <binding uri="xs:anyURI"? name="xs:QName"?
                     requires="list of xs:QName"?
                     policySets="list of xs:QName"?/>+
        </callback>
   </reference>

   <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
           many="xs:boolean"? mustSupply="xs:boolean"?>*
         default-property-value?
   </property>

   <component name="xs:NCName" autowire="xs:boolean"?
         requires="list of xs:QName"? policySets="list of xs:QName"?>*
         <implementation/>?
         <service name="xs:NCName" requires="list of xs:QName"?
               policySets="list of xs:QName"?>*
               <interface/>?
               <binding uri="xs:anyURI"? name="xs:QName"?
                     requires="list of xs:QName"?
                     policySets="list of xs:QName"?/>*
               <callback>?
                     <binding uri="xs:anyURI"? name="xs:QName"?
                           requires="list of xs:QName"?
                           policySets="list of xs:QName"?/>+
               </callback>
         </service>
         <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
               source="xs:string"? file="xs:anyURI"?>*
               property-value
         </property>
         <reference name="xs:NCName" target="list of xs:anyURI"?
               autowire="xs:boolean"? wiredByImpl="xs:boolean"?
               requires="list of xs:QName"? policySets="list of xs:QName"?
               multiplicity="0..1 or 1..1 or 0..n or 1..n"?/>*
               <interface/>?
               <binding uri="xs:anyURI"? name="xs:QName"?
                     requires="list of xs:QName"?
                        policySets="list of xs:QName"?/>*
               <callback>?
                     <binding uri="xs:anyURI"? name="xs:QName"?
                           requires="list of xs:QName"?
                           policySets="list of xs:QName"?/>+
         </callback>
         </reference>
   </component>

   <wire source="xs:anyURI" target="xs:anyURI" />*

</composite>
```

The composite element has the following ***attributes***:

- ***name (required)*** – the name of the composite. The form of a composite name is an XML QName, in the namespace identified by the targetNamespace attribute.
- ***targetNamespace (optional) –*** an identifier for a target namespace into which the composite is declared
- ***local (optional)*** – whether all the components within the composite must all run in the same operating system process. local="true" means that all the components must run in the same process. local="false", which is the default, means that different components within the composite may run in different operating system processes and they may even run on different nodes on a network.
- ***autowire (optional)*** – whether contained component references should be autowired, as described in the Autowire section. Default is false.
- ***constrainingType (optional)*** – the name of a constrainingType. When specified, the set of services, references and properties of the composite, plus related intents, is constrained to the set defined by the constrainingType. See the ConstrainingType Section for more details.
- ***requires (optional)*** – a list of policy intents. See the Policy Framework specification [10] for a description of this attribute.
- ***policySets (optional)*** – a list of policy sets. See the Policy Framework specification [10] for a description of this attribute.

Composites contain ***zero or more properties, services***, ***components***, ***references***, ***wires and included composites***. These artifacts are described in detail in the following sections.

Components contain configured implementations which hold the business logic of the composite. The components offer services and require references to other services. Composite services define the public services provided by the composite, which can be accessed from outside the composite. Composite references represent dependencies which the composite has on services provided elsewhere, outside the composite. Wires describe the connections between component services and component references within the composite. Included composites contribute the elements they contain to the using composite.

Composite services involve the ***promotion*** of one service of one of the components within the composite, which means that the composite service is actually provided by one of the components within the composite. Composite references involve the ***promotion*** of one or more references of one or more components. Multiple component references can be promoted to the same composite reference, as long as all the component references are compatible with one another. Where multiple component references are promoted to the same composite reference, then they all share the same configuration, including the same target service(s).

Composite services and composite references can use the configuration of their promoted services and references respectively (such as Bindings and Policy Sets). Alternatively composite services and composite references can override some or all of the configuration of the promoted services and references, through the configuration of bindings and other aspects of the composite service or reference.

Component services and component references can be promoted to composite services and references and also be wired internally within the composite at the same time. For a reference, this only makes sense if the reference supports a multiplicity greater than 1.

6.1 Property – Definition and Configuration

Properties allow for the configuration of an implementation with externally set data values. An implementation, including a composite, can declare zero or more properties. Each property has a type, which may be either simple or complex. An implementation may also define a default value for a property. Properties are configured with values in the components that use the implementation.

The declaration of a property in a composite follows the form described in the following schema snippet:

```
<?xml version="1.0" encoding="ASCII"?>

<composite     xmlns="http://www.osoa.org/xmlns/sca/1.0"
               name="xs:QCName" ... >

   ...

   <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
           many="xs:boolean"? mustSupply="xs:boolean"?>*
         default-property-value?
   </property>
   ...

</composite>
```

The property element has the following attributes:

- ***name (required)*** - the name of the property
- one of ***(required)***:
  - ***type*** – the type of the property - the qualified name of an XML schema type
  - ***element*** – the type of the property defined as the qualified name of an XML schema global element – the type is the type of the global element

- ***many (optional)*** - whether the property is single-valued (false) or multi-valued (true). The default is ***false***. In the case of a multi-valued property, it is presented to the implementation as a Collection of property values.
- ***mustSupply (optional)*** – whether the property value must be supplied by the component that uses the implementation – when mustSupply="true" the component must supply a value since the implementation has no default value for the property. A default-property-value should only be supplied when mustSupply="false" (the default setting for the mustSupply attribute), since the implication of a default value is that it is used only when a value is not supplied by the using component.

The property element may contain an optional ***default-property-value***, which provides default value for the property. The default value must match the type declared for the property:

- a string, if ***type*** is a simple type (must match the ***type*** declared)
- a complex type value matching the type declared by ***type***
- an element matching the element named by ***element***
- multiple values are permitted if many="true" is specified

Implementation types other than ***composite*** can declare properties in an implementation-dependent form (eg annotations within a Java class), or through a property declaration of exactly the form described above in a componentType file.

Property values can be configured when an implementation is used by a component. The form of the property configuration is shown in the section on Components.

6.1.1 Property Examples

For the following example of Property declaration and value setting, the following complex type is used as an example:

```
<xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
               targetNamespace="http://foo.com/"
               xmlns:tns="http://foo.com/">
   <!-- ComplexProperty schema -->
   <xsd:element name="fooElement" type="MyComplexType"/>
   <xsd:complexType name="MyComplexType">
         <xsd:sequence>
               <xsd:element name="a" type="xsd:string"/>
               <xsd:element name="b" type="anyURI"/>
         </xsd:sequence>
         <attribute name="attr" type="xsd:string" use="optional"/>
   </xsd:complexType>
</xsd:schema>
```

The following composite demostrates the declaration of a property of a complex type, with a default value, plus it demonstrates the setting of a property value of a complex type within a component:

```
<?xml version="1.0" encoding="ASCII"?>

<composite     xmlns="http://www.osoa.org/xmlns/sca/1.0"
               xmlns:foo="http://foo.com"
```

```
                  targetNamespace="http://foo.com"
                  name="AccountServices">
<!-- AccountServices Example1 -->

   ...

   <property name="complexFoo" type="foo:MyComplexType">
         <MyComplexPropertyValue xsi:type="foo:MyComplexType">
               <foo:a>AValue</foo:a>
               <foo:b>InterestingURI</foo:b>
         </MyComplexPropertyValue>
   </property>

   <component name="AccountServiceComponent">
         <implementation.java class="foo.AccountServiceImpl"/>
         <property name="complexBar" source="$complexFoo"/>
         <reference name="accountDataService"
               target="AccountDataServiceComponent"/>
         <reference name="stockQuoteService" target="StockQuoteService"/>
   </component>

   ...

</composite>
```

In the declaration of the property named ***complexFoo*** in the composite ***AccountServices***, the property is defined to be of type ***foo:MyComplexType***. The namespace ***foo*** is declared in the composite and it references the example XSD, where MyComplexType is defined. The declaration of complexFoo contains a default value. This is declared as the content of the property element. In this example, the default value consists of the element ***MyComplexPropertyValue*** of type foo:MyComplexType and its two child elements <foo:a> and <foo:b>, following the definition of MyComplexType.

In the component ***AccountServiceComponent***, the component sets the value of the property ***complexBar***, declared by the implementation configured by the component. In this case, the type of complexBar is foo:MyComplexType. The example shows that the value of the complexBar property is set from the value of the complexFoo property – the ***source*** attribute of the property element for complexBar declares that the value of the property is set from the value of a property of the containing composite. The value of the source attribute is ***$complexFoo***, where complexFoo is the name of a property of the composite. This value implies that the whole of the value of the source property is used to set the value of the component property.

The following example illustrates the setting of the value of a property of a simple type (a string) from ***part*** of the value of a property of the containing composite which has a complex type:

```
<?xml version="1.0" encoding="ASCII"?>

<composite        xmlns="http://www.osoa.org/xmlns/sca/1.0"
                  xmlns:foo="http://foo.com"
                  targetNamespace="http://foo.com"
                  name="AccountServices">
```

```
<!-- AccountServices Example2 -->

   ...

   <property name="complexFoo" type="foo:MyComplexType">
         <MyComplexPropertyValue xsi:type="foo:MyComplexType">
               <foo:a>AValue</foo:a>
               <foo:b>InterestingURI</foo:b>
         </MyComplexPropertyValue>
   </property>

   <component name="AccountServiceComponent">
         <implementation.java class="foo.AccountServiceImpl"/>
         <property name="currency" source="$complexFoo/a"/>
         <reference name="accountDataService"
               target="AccountDataServiceComponent"/>
         <reference name="stockQuoteService" target="StockQuoteService"/>
   </component>

   ...

</composite>
```

In this example, the component ***AccountServiceComponent*** sets the value of a property called ***currency***, which is of type string. The value is set from a property of the composite ***AccountServices*** using the source attribute set to ***$complexFoo/a***. This is an XPath expression that selects the property name ***complexFoo*** and then selects the value of the ***a*** subelement of complexFoo. The "a" subelement is a string, matching the type of the currency property.

Further examples of declaring properties and setting property values in a component follow:

Declaration of a property with a simple type and a default value:

```
<property name="SimpleTypeProperty" type="xsd:string">
MyValue
</property>
```

Declaration of a property with a complex type and a default value:

```
<property name="complexFoo" type="foo:MyComplexType">
  <MyComplexPropertyValue xsi:type="foo:MyComplexType">
     <foo:a>AValue</foo:a>
     <foo:b>InterestingURI</foo:b>
  </MyComplexPropertyValue>
</property>
```

Declaration of a property with an element type:

```
<property name="elementFoo" element="foo:fooElement">
  <foo:fooElement>
```

```
        <foo:a>AValue</foo:a>
        <foo:b>InterestingURI</foo:b>
    </foo:fooElement>
</property>
```

Property value for a simple type:

```
<property name="SimpleTypeProperty">
MyValue
</property>
```

Property value for a complex type, also showing the setting of an attribute value of the complex type:

```
<property name="complexFoo">
  <MyComplexPropertyValue xsi:type="foo:MyComplexType" attr="bar">
     <foo:a>AValue</foo:a>
     <foo:b>InterestingURI</foo:b>
  </MyComplexPropertyValue>
</property>
```

Property value for an element type:

```
<property name="elementFoo">
  <foo:fooElement attr="bar">
     <foo:a>AValue</foo:a>
     <foo:b>InterestingURI</foo:b>
  </foo:fooElement>
</property>
```

Declaration of a property with a complex type where multiple values are supported:

```
<property name="complexFoo" type="foo:MyComplexType" many="true"/>
```

Setting of a value for that property where multiple values are supplied:

```
<property name="complexFoo">
  <MyComplexPropertyValue1 xsi:type="foo:MyComplexType" attr="bar">
     <foo:a>AValue</foo:a>
     <foo:b>InterestingURI</foo:b>
  </MyComplexPropertyValue1>
  <MyComplexPropertyValue2 xsi:type="foo:MyComplexType" attr="zing">
     <foo:a>BValue</foo:a>
     <foo:b>BoringURI</foo:b>
  </MyComplexPropertyValue2>
</property>
```

6.2 References

The ***references of a composite*** are defined by ***promoting*** references defined by components contained in the composite. Each promoted reference indicates that the component reference must be resolved by services outside the composite. A component reference is promoted using a composite ***reference element***.

A composite reference is represented by a ***reference element*** which is a child of a composite element. There can be ***zero or more*** *reference* elements in a composite. The following snippet shows the composite schema with the schema for a ***reference*** element.

```
<?xml version="1.0" encoding="ASCII"?>
<!-- Reference schema snippet -->
<composite     xmlns="http://www.osoa.org/xmlns/sca/1.0"
               targetNamespace="xs:anyURI"
               name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?
               constrainingType="QName"?
               requires="list of xs:QName"? policySets="list of
xs:QName"?>

   ...

   <reference name="xs:NCName" target="list of xs:anyURI"?
         promote="list of xs:anyURI" wiredByImpl="xs:boolean"?
         multiplicity="0..1 or 1..1 or 0..n or 1..n"?
         requires="list of xs:QName"? policySets="list of xs:QName"?>*
         <interface/>?
         <binding uri="xs:anyURI"? name="xs:QName"?
               requires="list of xs:QName" policySets="list of
xs:QName"?/>*
         <callback>?
               <binding uri="xs:anyURI"? name="xs:QName"?
                     requires="list of xs:QName"?
                     policySets="list of xs:QName"?/>+
        </callback>
   </reference>

   ...

</composite>
```

The ***reference*** element has the following ***attributes***:

- ***name (required)*** – the name of the reference. The name must be unique across all the composite references in the composite. The name of the composite reference can be different then the name of the promoted component reference.
- ***promote (required)*** – identifies one or more promoted component references. The value is a list of values of the form <component-name>/<reference-name> separated by spaces. The specification of the reference name is optional if the component has only one reference.
- ***requires (optional)*** – a list of required policy intents. See the Policy Framework specification [10] for a description of this attribute.
- ***policySets (optional)*** – a list of policy sets. See the Policy Framework specification [10] for a description of this attribute.
- ***multiplicity (optional)*** - Defines the number of wires that can connect the reference to target services. The multiplicity can have the following values
  - 1..1 – one wire can have the reference as a source
  - 0..1 – zero or one wire can have the reference as a source
  - 1..n – one or more wires can have the reference as a source
  - 0..n - zero or more wires can have the reference as a source
- ***target (optional)*** – a list of one or more of target service URI's, depending on multiplicity setting. Each value wires the reference to a service in a composite that uses the composite containg the reference as an implementation for one of its components. For more details on wiring see the section on Wires.
- ***wiredByImpl (optional)*** – a boolean value, "false" by default, which indicates that the implementation wires this reference dynamically. If set to "true" it indicates that the target of the reference is set at runtime by the implementation code (eg by the code obtaining an endpoint reference by some means and setting this as the target of the reference through the use of programming interfaces defined by the relevant Client and Implementation specification). If "true" is set, then the reference should not be wired statically within a using composite, but left unwired.

The composite reference can optionally specify an ***interface***, ***multiplicity***, ***required intents*** , and ***bindings***. Whatever is not specified is defaulted from the promoted component reference(s).

If an ***interface*** is specified it must provide an interface which is the same or which is a compatible superset of the interface declared by the promoted component reference, i.e. provide a superset of the operations defined by the component for the reference. The interface is described by ***zero or one interface element*** which is a child element of the reference element. For details on the interface element see the Interface section.

The value specified for the ***multiplicity*** attribute has to be compatible with the multiplicity specified on the component reference, i.e. it has to be equal or further restrict. So a composite reference of multiplicity 0..1 or 1..1 can be used where the promoted component reference has multiplicity 0..n and 1..n respectively. However, a composite reference of multiplicity 0..n or 1..n cannot be used to promote a component reference of multiplicity 0..1 or 1..1 respectively.

Specified ***required intents*** add to or further qualify the required intents defined for the promoted component reference.

If one or more ***bindings*** are specified they ***override*** any and all of the bindings defined for the promoted component reference from the composite reference perspective. The bindings defined on the component reference are still in effect for local wires within the composite that have the component reference as their source. A reference element has zero or more ***binding elements*** as children. Details of the binding element are described in the Bindings section. For more details on wiring see the section on Wires.

Note that a binding element may specify an endpoint which is the target of that binding. A reference must not mix the use of endpoints specified via binding elements with target endpoints specified via the target attribute. If the target attribute is set, then binding elements can only list one or more binding types that can be used for the wires identified by the target attribute. All the binding types identified are available for use on each wire in this case. If endpoints are specified in the binding elements, each endpoint must use the binding type of the binding element in which it is defined. In addition, each binding element needs to specify an endpoint in this case.

A ***reference*** element has an optional ***callback*** element used if the interface has a callback defined, which has one or more ***binding*** elements as children. The ***callback*** and its binding child elements are specified if there is a need to have binding details used to handle callbacks. If the callback element is not present, the behaviour is runtime implementation dependent.

The same component reference maybe promoted more than once, using different composite references, but only if the multiplicity defined on the component reference is 0..n or 1..n. The multiplicity on the composite reference can restrict accordingly.

Two or more component references may be promoted by one composite reference, but only when

- the interfaces of the component references are the same, or if the composite reference itself declares an interface then all the component references must have interfaces which are compatible with the composite reference interface
- the multiplicities of the component references are compatible, i.e one can be the restricted form of the another, which also means that the composite reference carries the restricted form either implicitly or explicitly
- the intents declared on the component references must be compatible – the intents which apply to the composite reference in this case are the union of the required intents specified for each of the promoted component references. If any intents contradict (eg mutually incompatible qualifiers for a particular intent) then there is an error.

6.2.1 Example Reference

The following figure shows the reference symbol that is used to represent a reference in an assembly diagram.

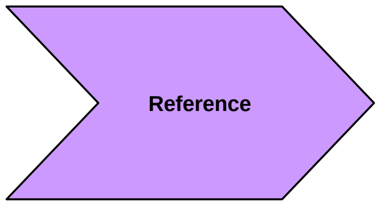

Figure 7: Reference symbol

The following figure shows the assembly diagram for the MyValueComposite containing the reference CustomerService and the reference StockQuoteService.

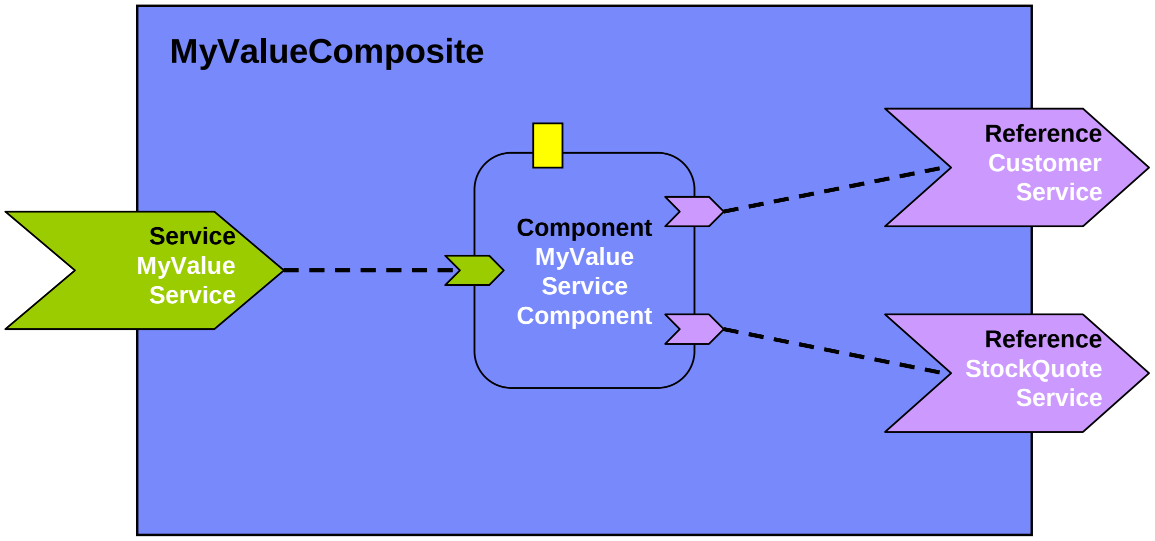
MyValueComposite
Reference Customer Service
Service MyValue Service
Component MyValue Service Component
Reference StockQuote Service

Figure 8: MyValueComposite showing References

The following snippet shows the MyValueComposite.composite file for the MyValueComposite containing the reference elements for the CustomerService and the StockQuoteService. The reference CustomerService is bound using the SCA binding. The reference StockQuoteService is bound using the Web service binding. The endpoint addresses of the bindings can be specified, for example using the binding ***uri*** attribute (for details see the Bindings section), or overridden in an enclosing composite. Although in this case the reference StockQuoteService is bound to a Web service, its interface is defined by a Java interface, which was created from the WSDL portType of the target web service.

```
<?xml version="1.0" encoding="ASCII"?>
<!-- MyValueComposite_3 example -->
<composite     xmlns="http://www.osoa.org/xmlns/sca/1.0"
               targetNamespace="http://foo.com"
               name="MyValueComposite" >

   ...

   <component name="MyValueServiceComponent">
         <implementation.java
class="services.myvalue.MyValueServiceImpl"/>
         <property name="currency">EURO</property>
         <reference name="customerService"/>
         <reference name="StockQuoteService"/>
   </component>

   <reference name="CustomerService"
         promote="MyValueServiceComponent/customerService">
         <interface.java interface="services.customer.CustomerService"/>
         <!-- The following forces the binding to be binding.sca whatever
is -->
```

```
            <!-- specified by the component reference or by the underlying
-->
            <!-- implementation
   -->
            <binding.sca/>
      </reference>

      <reference name="StockQuoteService"
            promote="MyValueServiceComponent/StockQuoteService">
            <interface.java
   interface="services.stockquote.StockQuoteService"/>
            <binding.ws port="http://www.stockquote.org/StockQuoteService#

   wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
      </reference>

      ...

   </composite>
```

6.3 Service

The ***services of a composite*** are defined by promoting services defined by components contained in the composite. A component service is promoted by means of a composite ***service element***.

A composite service is represented by a ***service element*** which is a child of the composite element. There can be ***zero or more*** service elements in a composite. The following snippet shows the composite schema with the schema for a service child element:

```
<?xml version="1.0" encoding="ASCII"?>
<!-- Servicee schema snippet -->
<composite     xmlns="http://www.osoa.org/xmlns/sca/1.0"
               targetNamespace="xs:anyURI"
               name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?
               constrainingType="QName"?
               requires="list of xs:QName"? policySets="list of
xs:QName"?>
   ...

   <service name="xs:NCName" promote="xs:anyURI"
         requires="list of xs:QName"? policySets="list of xs:QName"?>*
         <interface/>?
         <binding uri="xs:anyURI"? name="xs:QName"?
               requires="list of xs:QName" policySets="list of
xs:QName"?/>*
         <callback>?
```

```
                  <binding uri="xs:anyURI"? name="xs:QName"?
                        requires="list of xs:QName"?
                        policySets="list of xs:QName"?/>+
           </callback>
      </service>

      ...

   </composite>
```

The service element has the following ***attributes***:

- ***name (required)*** – the name of the service, the name MUST BE unique across all the composite services in the composite. The name of the composite service can be different from the name of the promoted component service.
- ***promote (required)*** – identifies the promoted service, the value is of the form <component-name>/<service-name>. The service name is optional if the target component only has one service.
- ***requires (optional)*** – a list of required policy intents. See the Policy Framework specification [10] for a description of this attribute.
- ***policySets (optional)*** – a list of policy sets. See the Policy Framework specification [10] for a description of this attribute.

The composite service can optionally specify an ***interface***, ***required intents*** and ***bindings***. Whatever is not specified is defaulted from the promoted component service.

If an ***interface*** is specified it must be the same or a compatible subset of the interface provided by the promoted component service, i.e. provide a subset of the operations defined by the component service. The interface is described by ***zero or one interface element*** which is a child element of the service element. For details on the interface element see the Interface section.

Specified ***required intents*** add to or further qualify the required intents defined by the promoted component service.

If bindings are specified they ***override*** the bindings defined for the promoted component service from the composite service perspective. The bindings defined on the component service are still in effect for local wires within the composite that target the component service. A service element has zero or more ***binding elements*** as children. Details of the binding element are described in the Bindings section. For more details on wiring see the Wiring section.

A service element has an optional ***callback*** element used if the interface has a callback defined,, which has one or more ***binding*** elements as children. The ***callback*** and its binding child elements are specified if there is a need to have binding details used to handle callbacks. If the callback element is not present, the behaviour is runtime implementation dependent.

The same component service can be promoted by more then one composite service.

6.3.1 Service Examples

The following figure shows the service symbol that used to represent a service in an assembly diagram:

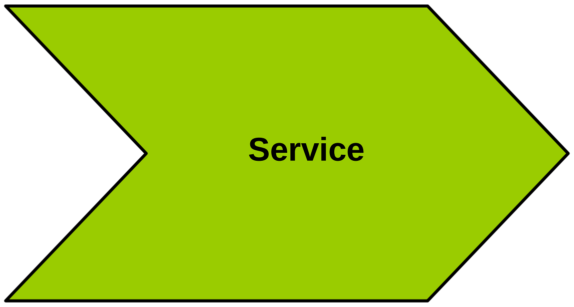

Figure 9: Service symbol

The following figure shows the assembly diagram for the MyValueComposite containing the service MyValueService.

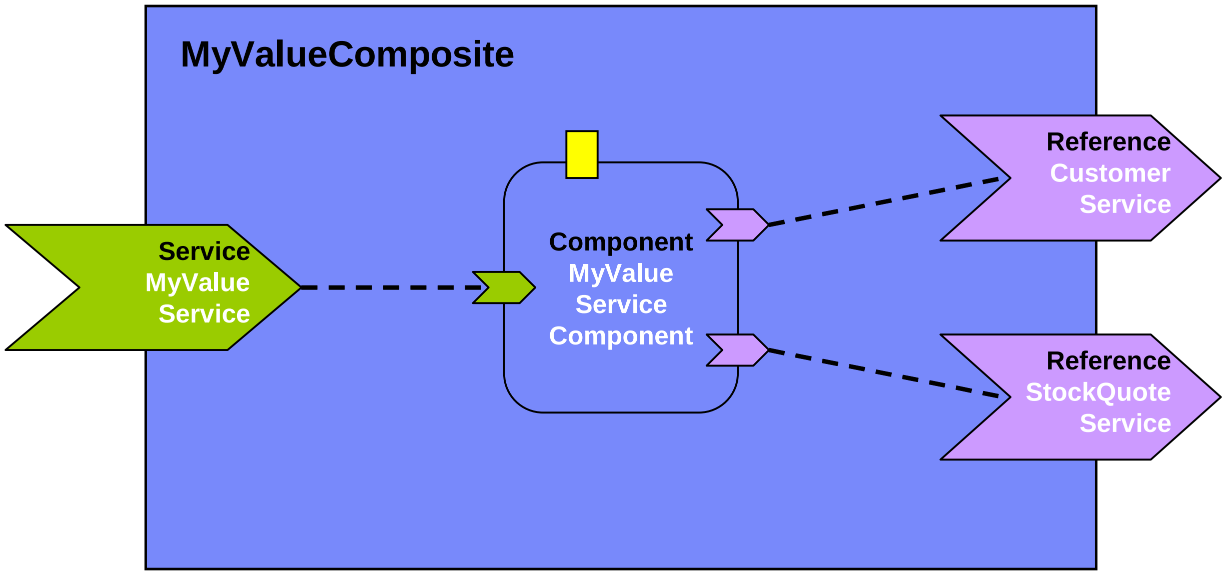

Figure 10: MyValueComposite showing Service

The following snippet shows the MyValueComposite.composite file for the MyValueComposite containing the service element for the MyValueService, which is a promote of the service offered by the MyValueServiceComponent. The name of the promoted service is omitted since MyValueServiceComponent offers only one service. The composite service MyValueService is bound using a Web service binding.

```
<?xml version="1.0" encoding="ASCII"?>
<!-- MyValueComposite_4 example -->
<composite     xmlns="http://www.osoa.org/xmlns/sca/1.0"
               targetNamespace="http://foo.com"
               name="MyValueComposite" >
```

```
   ...

   <service name="MyValueService" promote="MyValueServiceComponent">
         <interface.java interface="services.myvalue.MyValueService"/>
         <binding.ws port="http://www.myvalue.org/MyValueService#
            wsdl.endpoint(MyValueService/MyValueServiceSOAP)"/>
   </service>

   <component name="MyValueServiceComponent">
         <implementation.java
class="services.myvalue.MyValueServiceImpl"/>
         <property name="currency">EURO</property>
         <service name="MyValueService"/>
         <reference name="customerService"/>
         <reference name="StockQuoteService"/>
   </component>

   ...

</composite>
```

6.4 Wire

SCA wires within a composite connect ***source component references*** to ***target component services***.

One way of defining a wire is by ***configuring a reference of a component using its target attribute***. The reference element is configured with the wire-target-URI of the service(s) that resolve the reference. Multiple target services are valid when the reference has a multiplicity of 0..n or 1..n.

An alternative way of defining a Wire is by means of a ***wire element*** which is a child of the composite element. There can be ***zero or more*** wire elements in a composite. This alternative method for defining wires is useful in circumstances where separation of the wiring from the elements the wires connect helps simplify development or operational activities. An example is where the components used to build a domain are relatively static but where new or changed applications are created regularly from those components, through the creation of new assemblies with different wiring. Deploying the wiring separately from the components allows the wiring to be created or modified with minimum effort.

Note that a Wire specified via a wire element is equivalent to a wire specified via the target attribute of a reference. The rule which forbids mixing of wires specified with the target attribute with the specification of endpoints in binding subelements of the reference also applies to wires specified via separate wire elements.

The following snippet shows the composite schema with the schema for the reference elements of components and composite services and the wire child element:

```
<?xml version="1.0" encoding="ASCII"?>
<!-- Wires schema snippet -->
<composite     xmlns="http://www.osoa.org/xmlns/sca/1.0"
```

```
                    targetNamespace="xs:anyURI"
                    name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?
                    constrainingType="QName"?
                    requires="list of xs:QName"? policySets="list of
xs:QName"?>

    ...

    <wire source="xs:anyURI" target="xs:anyURI" />*

</composite>
```

The ***reference element of a component*** and the ***reference element of a service*** has a list of one or more of the following ***wire-target-URI*** values for the target, with multiple values separated by a space:

- <component-name>/<service-name>
    - where the target is a service of a component. The specification of the service name is optional if the target component only has one service with a compatible interface

The ***wire element*** has the following attributes:

- ***source (required)*** – names the source component reference. Valid URI schemes are:
    - <component-name>/<reference-name>
        - where the source is a component reference. The specification of the reference name is optional if the source component only has one reference
- ***target (required)*** – names the target component service. Valid URI schemes are
    - <component-name>/<service-name>
        - where the target is a service of a component. The specification of the service name is optional if the target component only has one service with a compatible interface

For a composite used as a component implementation, wires can only link sources and targets that are contained in the same composite (irrespective of which file or files are used to describe the composite). Wiring to entities outside the composite is done through services and references of the composite with wiring defined by the next higher composite.

A wire may only connect a source to a target if the target implements an interface that is compatible with the interface required by the source**.** The source and the target are compatible if:

1. the source interface and the target interface MUST either both be remotable or they are both local
2. the operations on the target interface MUST be the same as or be a superset of the operations in the interface specified on the source
3. compatibility for the individual operation is defined as compatibility of the signature, that is operation name, input types, and output types MUST BE the same.
4. the order of the input and output types also MUST BE the same.
5. the set of Faults and Exceptions expected by the source MUST BE the same or be a superset of those specified by the target.

6. other specified attributes of the two interfaces MUST match, including Scope and Callback interface

A Wire can connect between different interface languages (eg. Java interfaces and WSDL portTypes) in either direction, as long as the operations defined by the two interface types are equivalent. They are equivalent if the operation(s), parameter(s), return value(s) and faults/exceptions map to each other.

Service clients cannot (portably) ask questions at runtime about additional interfaces that are provided by the implementation of the service (e.g. the result of "instance of" in Java is non portable). It is valid for an SCA implementation to have proxies for all wires, so that, for example, a reference object passed to an implementation may only have the business interface of the reference and may not be an instance of the (Java) class which is used to implement the target service, even where the interface is local and the target service is running in the same process.

Note: It is permitted to deploy a composite that has references that are not wired. For the case of an un-wired reference with multiplicity 1..1 or 1..n the deployment process provided by an SCA runtime SHOULD issue a warning.

6.4.1 Wire Examples

The following figure shows the assembly diagram for the MyValueComposite2 containing wires between service, components and references.

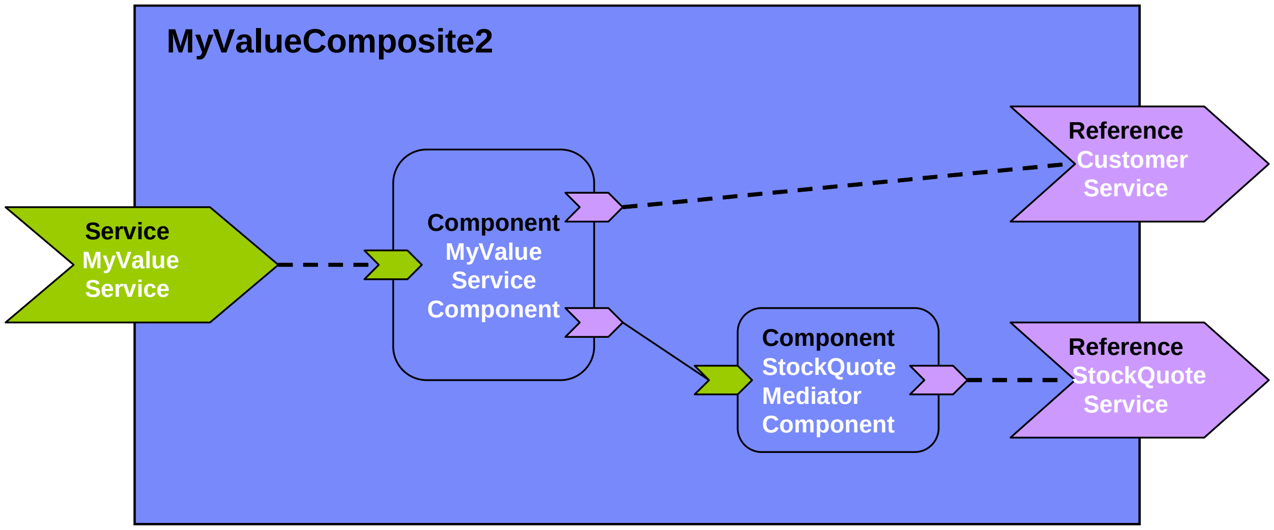
MyValueComposite2
Service MyValue Service
Component MyValue Service Component
Reference Customer Service
Component StockQuote Mediator Component
Reference StockQuote Service

Figure 11: MyValueComposite2 showing Wires

The following snippet shows the MyValueComposite2.composite file for the MyValueComposite2 containing the configured component and service references. The service MyValueService is wired to the MyValueServiceComponent. The MyValueServiceComponent's customerService reference is wired to the composite's CustomerService reference. The MyValueServiceComponent's stockQuoteService reference is wired to the StockQuoteMediatorComponent, which in turn has its reference wired to the StockQuoteService reference of the composite.

```
<?xml version="1.0" encoding="ASCII"?>
<!-- MyValueComposite Wires examples -->
<composite     xmlns="http://www.osoa.org/xmlns/sca/1.0"
               targetNamespace="http://foo.com"
```

```
                    name="MyValueComposite2" >

   <service name="MyValueService" promote="MyValueServiceComponent">
         <interface.java interface="services.myvalue.MyValueService"/>
         <binding.ws port="http://www.myvalue.org/MyValueService#
               wsdl.endpoint(MyValueService/MyValueServiceSOAP)"/>
   </service>

   <component name="MyValueServiceComponent">
         <implementation.java
class="services.myvalue.MyValueServiceImpl"/>
         <property name="currency">EURO</property>
         <service name="MyValueService"/>
         <reference name="customerService"/>
         <reference name="stockQuoteService"
               target="StockQuoteMediatorComponent"/>
   </component>

   <component name="StockQuoteMediatorComponent">
         <implementation.java class="services.myvalue.SQMediatorImpl"/>
         <property name="currency">EURO</property>
         <reference name="stockQuoteService"/>
   </component>

   <reference name="CustomerService"
         promote="MyValueServiceComponent/customerService">
         <interface.java interface="services.customer.CustomerService"/>
         <binding.sca/>
   </reference>

   <reference name="StockQuoteService"
promote="StockQuoteMediatorComponent">
         <interface.java
interface="services.stockquote.StockQuoteService"/>
         <binding.ws port="http://www.stockquote.org/StockQuoteService#
               wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
   </reference>

</composite>
```

6.4.2 Autowire

SCA provides a feature named ***Autowire***, which can help to simplify the assembly of composites. Autowire enables component references to be automatically wired to component services which will satisfy those references, without the need to create explicit wires between the references and the services. When the autowire feature is used, a component reference which is not promoted and which is not explicitly wired to a service within a composite is automatically wired to a target service within the same composite. Autowire works by searching within the composite for a service interface which matches the interface of the references.

The autowire feature is not used by default. Autowire is enabled by the setting of an autowire attribute to "true". Autowire is disabled by setting of the autowire attribute to "false" The autowire attribute can be applied to any of the following elements within a composite:

- reference
- component
- composite

Where an element does not have an explicit setting for the autowire attribute, it inherits the setting from its parent element. Thus a reference element inherits the setting from its containing component. A component element inherits the setting from its containing composite. Where there is no setting on any level, autowire="false" is the default.

As an example, if a composite element has autowire="true" set, this means that autowiring is enabled for all component references within that composite. In this example, autowiring can be turned off for specific components and specific references through setting autowire="false" on the components and references concerned.

For each component reference for which autowire is enabled, the autowire process searches within the composite for target services which are compatible with the reference. "Compatible" here means:

- the target service interface must be a compatible superset of the reference interface (as defined in the section on Wires)
- the intents, bindings and policies applied to the service must be compatible on the reference – so that wiring the reference to the service will not cause an error due to binding and policy mismatch (see the Policy Framework specification [10] for details)

If the search finds ***more than 1*** valid target service for a particular reference, the action taken depends on the multiplicity of the reference:

- for multiplicity 0..1 and 1..1, the SCA runtime selects one of the target services in a runtime-dependent fashion and wires the reference to that target service
- for multiplicity 0..n and 1..n, the reference is wired to all of the target services

If the search finds ***no*** valid target services for a particular reference, the action taken depends on the multiplicy of the reference:

- for multiplicity 0..1 and 0..n, there is no problem – no services are wired and there is no error
- for multiplicity 1..1 and 1..n, an error is raised by the SCA runtime since the reference is intended to be wired

6.4.3 Autowire Examples

This example demonstrates two versions of the same composite – the first version is done using explicit wires, with no autowiring used, the second version is done using autowire. In both cases the end result is the same – the same wires connect the references to the services.

First, here is a diagram for the composite:

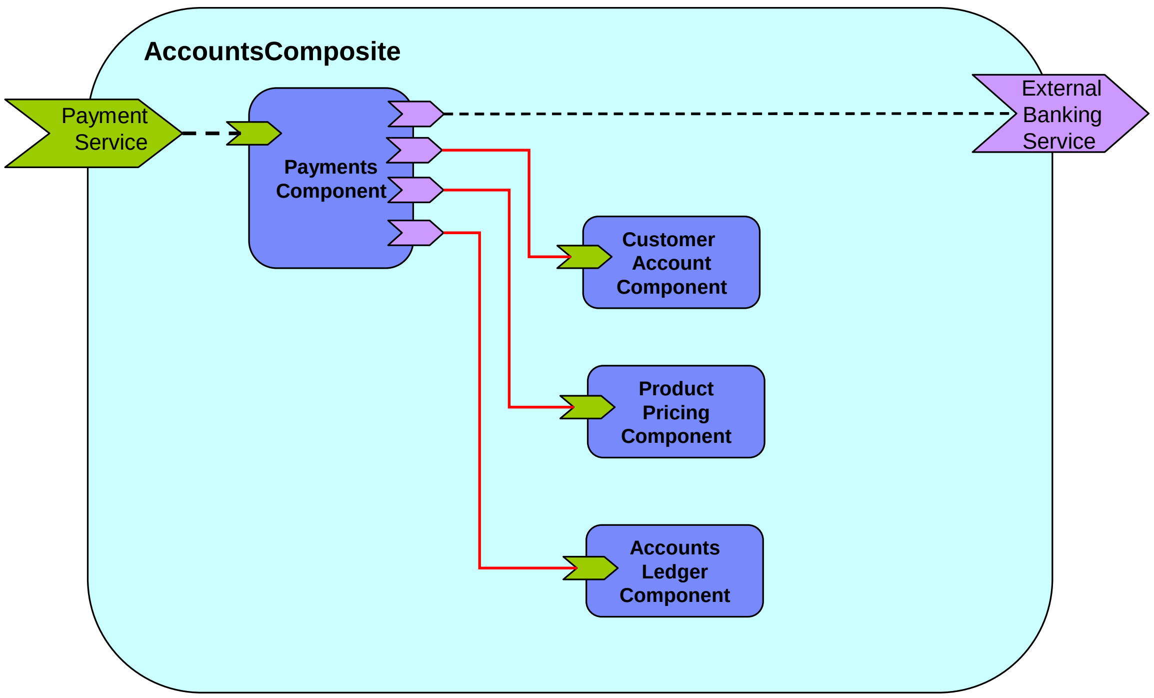
AccountsComposite
Payment Service
Payments Component
External Banking Service
Customer Account Component
Product Pricing Component
Accounts Ledger Component

Figure 12: Example Composite for Autowire

First, the composite using explicit wires:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Autowire Example - No autowire  -->
<composite  xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.osoa.org/xmlns/sca/1.0"
    targetNamespace="http://foo.com"
    name="AccountComposite">

    <service name="PaymentService" promote="PaymentsComponent"/>

    <component name="PaymentsComponent">
        <implementation.java class="com.foo.accounts.Payments"/>
        <service name="PaymentService"/>
        <reference name="CustomerAccountService"
           target="CustomerAccountComponent"/>
        <reference name="ProductPricingService"
target="ProductPricingComponent"/>
        <reference name="AccountsLedgerService"
target="AccountsLedgerComponent"/>
        <reference name="ExternalBankingService"/>
    </component>

    <component name="CustomerAccountComponent">
        <implementation.java class="com.foo.accounts.CustomerAccount"/>
    </component>

    <component name="ProductPricingComponent">
        <implementation.composite class="com.foo.accounts.ProductPricing"/>
    </component>

    <component name="AccountsLedgerComponent">
        <implementation.composite class="com.foo.accounts.AccountsLedger"/>
    </component>

    <reference name="ExternalBankingService"
        promote="PaymentsComponent/ExternalBankingService"/>

</composite>
```

Secondly, the composite using autowire:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Autowire Example - With autowire -->
<composite  xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.osoa.org/xmlns/sca/1.0"
    targetNamespace="http://foo.com"
    name="AccountComposite">

    <service name="PaymentService" promote="PaymentsComponent">
         <interface.java class="com.foo.PaymentServiceInterface"/>
    </service>

    <component name="PaymentsComponent" autowire="true">
        <implementation.java class="com.foo.accounts.Payments"/>
        <service name="PaymentService"/>
        <reference name="CustomerAccountService"/>
        <reference name="ProductPricingService"/>
        <reference name="AccountsLedgerService"/>
        <reference name="ExternalBankingService"/>
    </component>

    <component name="CustomerAccountComponent">
        <implementation.java class="com.foo.accounts.CustomerAccount"/>
    </component>

    <component name="ProductPricingComponent">
        <implementation.composite
class="com.foo.accounts.ProductPricing"/>
```

```
        </component>

        <component name="AccountsLedgerComponent">
            <implementation.composite
    class="com.foo.accounts.AccountsLedger"/>
        </component>

        <reference name="ExternalBankingService"
            promote="PaymentsComponent/ExternalBankingService"/>

    </composite>
```

In this second case, autowire is set on for the PaymentsComponent and there are no explicit wires for any of its references – the wires are created automatically through autowire.

Note: In the second example, it would be possible to omit all of the service and reference elements from the PaymentsComponent. They are left in for clarity, but if they are omitted, the component service and references still exist, since they are provided by the implementation used by the component.

6.5 Using Composites as Component Implementations

Composites may form ***component implementations*** in higher-level composites – in other words the higher-level composites can have components which are implemented by composites.

When a composite is used as a component implementation, it defines a boundary of visibility. Components within the composite cannot be referenced directly by the using component. The using component can only connect wires to the services and references of the used composite and set values for any properties of the composite. The internal construction of the composite is invisible to the using component.

A composite used as a component implementation must also honor a ***completeness contract***. The services, references and properties of the composite form a contract which is relied upon by the using component. The concept of completeness of the composite implies:

- the composite must have at least one service or at least one reference.
  A component with no services and no references is not meaningful in terms of SCA, since it cannot be wired to anything – it neither provides nor consumes any services

- each service offered by the composite must be wired to a service of a component or to a composite reference.
  If services are left unwired, the implication is that some exception will occur at runtime if the service is invoked.

The component type of a composite is defined by the set of service elements, reference elements and property elements that are the children of the composite element.

Composites are used as component implementations through the use of the ***implementation.composite*** element as a child element of the component. The schema snippet for the implementation.composite element is:

```

<?xml version="1.0" encoding="ASCII"?>
<!-- Composite Implementation schema snippet -->
<composite     xmlns="http://www.osoa.org/xmlns/sca/1.0"
               targetNamespace="xs:anyURI"
```

```
                        name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?
                        constrainingType="QName"?
                        requires="list of xs:QName"? policySets="list of
xs:QName"?>

   ...

   <component name="xs:NCName" autowire="xs:boolean"?
         requires="list of xs:QName"? policySets="list of xs:QName"?>*
         <implementation.composite name="xs:QName"/>?
         <service name="xs:NCName" requires="list of xs:QName"?
               policySets="list of xs:QName"?>*
               <interface/>?
               <binding uri="xs:anyURI" name="xs:QName"?
                     requires="list of xs:QName"
                     policySets="list of xs:QName"?/>*
               <callback>?
                     <binding uri="xs:anyURI"? name="xs:QName"?
                           requires="list of xs:QName"?
                           policySets="list of xs:QName"?/>+
         </callback>
         </service>
         <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
               source="xs:string"? file="xs:anyURI"?>*
               property-value
         </property>
         <reference name="xs:NCName" target="list of xs:anyURI"?
               autowire="xs:boolean"? wiredByImpl="xs:boolean"?
               requires="list of xs:QName"? policySets="list of xs:QName"?
               multiplicity="0..1 or 1..1 or 0..n or 1..n"?/>*
               <interface/>?
               <binding uri="xs:anyURI"? name="xs:QName"?
                     requires="list of xs:QName" policySets="list of
xs:QName"?/>*
               <callback>?
                     <binding uri="xs:anyURI"? name="xs:QName"?
                           requires="list of xs:QName"?
                           policySets="list of xs:QName"?/>+
         </callback>
         </reference>
   </component>

   ...
```

```
</composite>
```

The implementation.composite element has the following attribute:

- ***name (required)*** – the name of the composite used as an implementation

6.5.1 Example of Composite used as a Component Implementation

The following in an example of a composite which contains two components, each of which is implemented by a composite:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- CompositeComponent example -->
<composite  xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
    xsd:schemaLocation="http://www.osoa.org/xmlns/sca/1.0
file:/C:/Strategy/SCA/v09_osoaschemas/schemas/sca.xsd"
    xmlns="http://www.osoa.org/xmlns/sca/1.0"
    targetNamespace="http://foo.com"
    xmlns:foo="http://foo.com"
    name="AccountComposite">

    <service name="AccountService" promote="AccountServiceComponent">
        <interface.java interface="services.account.AccountService"/>
        <binding.ws port="AccountService#
            wsdl.endpoint(AccountService/AccountServiceSOAP)"/>
    </service>

    <reference name="stockQuoteService"
         promote="AccountServiceComponent/StockQuoteService">
        <interface.java
interface="services.stockquote.StockQuoteService"/>
        <binding.ws
port="http://www.quickstockquote.com/StockQuoteService#
            wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
    </reference>

    <property name="currency" type="xsd:string">EURO</property>

    <component name="AccountServiceComponent">
        <implementation.composite name="foo:AccountServiceComposite1"/>

        <reference name="AccountDataService" target="AccountDataService"/>
```

```
            <reference name="StockQuoteService"/>

            <property name="currency" source="$currency"/>
        </component>

        <component name="AccountDataService">
            <implementation.composite name="foo:AccountDataServiceComposite"/>

            <property name="currency" source="$currency"/>
        </component>

    </composite>
```

6.6 Using Composites through Inclusion

In order to assist team development, composites may be developed in the form of multiple physical artifacts that are merged into a single logical unit.

A composite is defined in an ***xxx.composite*** file and the composite may receive additional content through the ***inclusion of other composite*** files.

The semantics of included composites are that the content of the included composite is inlined into the using composite **xxx.composite** file through ***include*** elements in the using composite. The effect is one of ***textual inclusion*** – that is, the text content of the included composite is placed into the using composite in place of the include statement. The included composite element itself is discarded in this process – only its contents are included.

The composite file used for inclusion can have any contents, but always contains a single ***composite*** element. The composite element may contain any of the elements which are valid as child elements of a composite element, namely components, services, references, wires and includes. There is no need for the content of an included composite to be complete, so that artifacts defined within the using composite or in another associated included composite file may be referenced. For example, it is permissible to have two components in one composite file while a wire specifying one component as the source and the other as the target can be defined in a second included composite file.

It is an error if the (using) composite resulting from the inclusion is invalid – for example, if there are duplicated elements in the using composite (eg. two services with the same uri contributed by different included composites), or if there are wires with non-existent source or target.

The following snippet shows the partial schema for the include element.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Include snippet -->
<composite     xmlns="http://www.osoa.org/xmlns/sca/1.0"
               targetNamespace="xs:anyURI"
               name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?
               constrainingType="QName"?
               requires="list of xs:QName"? policySets="list of
xs:QName"?>

   ...
```

```
        <include name="xs:QName"/>*

        ...

    </composite>
```

The include element has the following ***attribute***:

- ***name (required)*** – the name of the composite that is included.

6.6.1 Included Composite Examples

The following figure shows the assembly diagram for the MyValueComposite2 containing four included composites. The ***MyValueServices composite*** contains the MyValueService service. The ***MyValueComponents composite*** contains the MyValueServiceComponent and the StockQuoteMediatorComponent as well as the wire between them. The ***MyValueReferences composite*** contains the CustomerService and StockQuoteService references. The ***MyValueWires composite*** contains the wires that connect the MyValueService service to the MyValueServiceComponent, that connect the customerService reference of the MyValueServiceComponent to the CustomerService reference, and that connect the stockQuoteService reference of the StockQuoteMediatorComponent to the StockQuoteService reference. Note that this is just one possible way of building the MyValueComposite2 from a set of included composites.

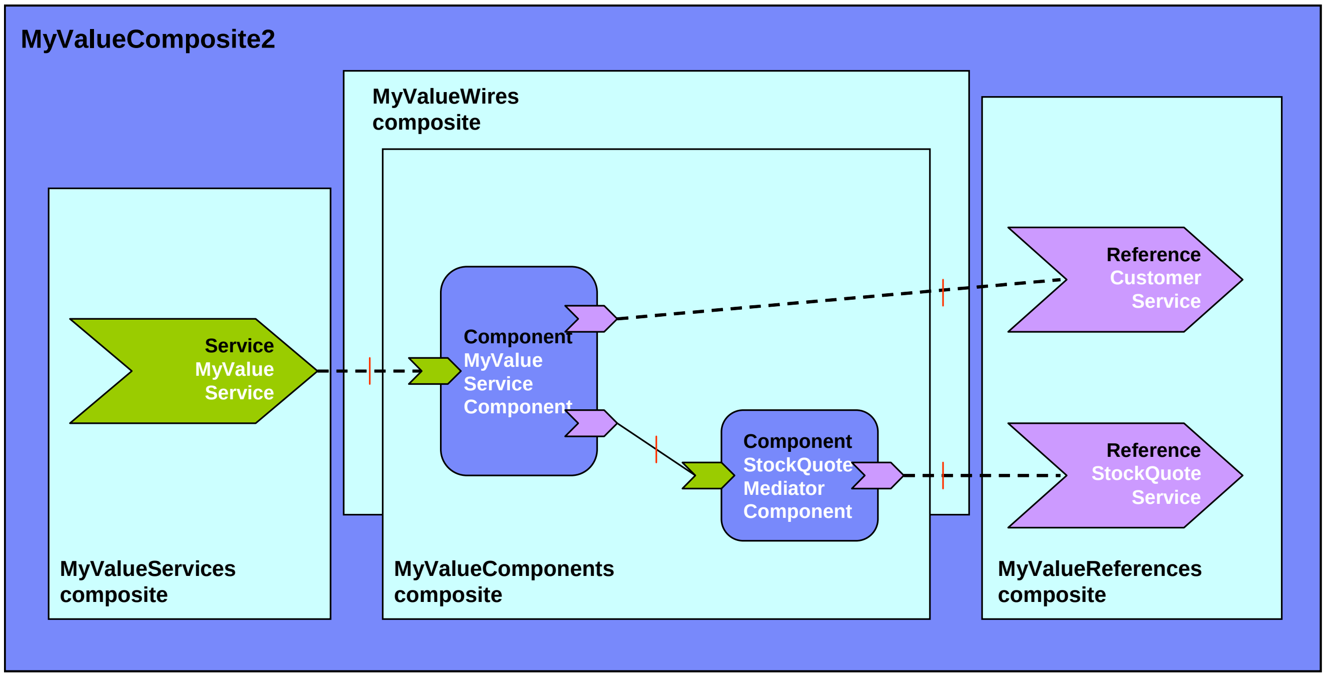
MyValueComposite2
MyValueWires composite
Service MyValue Service
Component MyValue Service Component
Component StockQuote Mediator Component
Reference Customer Service
Reference StockQuote Service
MyValueServices composite
MyValueComponents composite
MyValueReferences composite

Figure 13 MyValueComposite2 built from 4 included composites

The following snippet shows the contents of the MyValueComposite2.composite file for the MyValueComposite2 built using included composites. In this sample it only provides the name of the composite. The composite file itself could be used in a scenario using included composites to define components, services, references and wires.

```
<?xml version="1.0" encoding="ASCII"?>
<composite     xmlns="http://www.osoa.org/xmlns/sca/1.0"
               targetNamespace="http://foo.com"
               xmlns:foo="http://foo.com"
               name="MyValueComposite2" >

   <include name="foo:MyValueServices"/>
   <include name="foo:MyValueComponents"/>
   <include name="foo:MyValueReferences"/>
   <include name="foo:MyValueWires"/>

</composite>
```

The following snippet shows the content of the MyValueServices.composite file.

```
<?xml version="1.0" encoding="ASCII"?>
<composite     xmlns="http://www.osoa.org/xmlns/sca/1.0"
               targetNamespace="http://foo.com"
               xmlns:foo="http://foo.com"
               name="MyValueServices" >

   <service name="MyValueService" promote="MyValueServiceComponent">
         <interface.java interface="services.myvalue.MyValueService"/>
         <binding.ws port="http://www.myvalue.org/MyValueService#
               wsdl.endpoint(MyValueService/MyValueServiceSOAP)"/>
   </service>

</composite>
```

The following snippet shows the content of the MyValueComponents.composite file.

```
<?xml version="1.0" encoding="ASCII"?>
<composite     xmlns="http://www.osoa.org/xmlns/sca/1.0"
               targetNamespace="http://foo.com"
               xmlns:foo="http://foo.com"
               name="MyValueComponents" >

   <component name="MyValueServiceComponent">
         <implementation.java
class="services.myvalue.MyValueServiceImpl"/>
```

```
            <property name="currency">EURO</property>
      </component>

      <component name="StockQuoteMediatorComponent">
            <implementation.java class="services.myvalue.SQMediatorImpl"/>
            <property name="currency">EURO</property>
      </component>

<composite>
```

The following snippet shows the content of the MyValueReferences.composite file.

```
<?xml version="1.0" encoding="ASCII"?>
<composite     xmlns="http://www.osoa.org/xmlns/sca/1.0"
               targetNamespace="http://foo.com"
               xmlns:foo="http://foo.com"
               name="MyValueReferences" >

      <reference name="CustomerService"
            promote="MyValueServiceComponent/CustomerService">
            <interface.java interface="services.customer.CustomerService"/>
            <binding.sca/>
      </reference>

      <reference name="StockQuoteService"
promote="StockQuoteMediatorComponent">
            <interface.java
interface="services.stockquote.StockQuoteService"/>
            <binding.ws port="http://www.stockquote.org/StockQuoteService#
                  wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
      </reference>

</composite>
```

The following snippet shows the content of the MyValueWires.composite file.

```
<?xml version="1.0" encoding="ASCII"?>
<composite     xmlns="http://www.osoa.org/xmlns/sca/1.0"
               targetNamespace="http://foo.com"
               xmlns:foo="http://foo.com"
               name="MyValueWires" >

      <wire source="MyValueServiceComponent/stockQuoteService"
            target="StockQuoteMediatorComponent"/>
```

```
</composite>
```

6.7 Composites which Include Component Implementations of Multiple Types

A Composite containing multiple components MAY have multiple component implementation types. For example, a Composite may include one component with a Java POJO as its implementation and another component with a BPEL process as its implementation.

6.8 ConstrainingType

SCA allows a component, and its associated implementation, to be constrained by a ***constrainingType***. The constrainingType element provides assistance in developing top-down usecases in SCA, where an architect or assembler can define the structure of a composite, including the required form of component implementations, before any of the implementations are developed.

A constrainingType is expressed as an element which has services, reference and properties as child elements and which can have intents applied to it. The constrainingType is independent of any implementation. Since it is independent of an implementation it cannot contain any implementation-specific configuration information or defaults. Specifically, it cannot contain bindings, policySets, property values or default wiring information. The constrainingType is applied to a component through a constrainingType attribute on the component.

A constrainingType provides the "shape" for a component and its implementation. Any component configuration that points to a constrainingType is constrained by this shape. The constrainingType specifies the services, references and properties that must be implemented. This provides the ability for the implementer to program to a specific set of services, references and properties as defined by the constrainingType. Components are therefore configured instances of implementations and are constrained by an associated constrainingType.

If the configuration of the component or its implementation do not conform to the constrainingType, it is an error.

A constrainingType is represented by a ***constrainingType*** element. The following snippet shows the pseudo-schema for the composite element.

```
<?xml version="1.0" encoding="ASCII"?>
<!-- ConstrainingType schema snippet -->
<constrainingType    xmlns="http://www.osoa.org/xmlns/sca/1.0"
               targetNamespace="xs:anyURI"?
               name="xs:NCName" requires="list of xs:QName"?>


   <service name="xs:NCName" requires="list of xs:QName"?>*
         <interface/>?
   </service>

   <reference name="xs:NCName"
         multiplicity="0..1 or 1..1 or 0..n or 1..n"?
         requires="list of xs:QName"?>*
```

```
            <interface/>?
        </reference>

        <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
                many="xs:boolean"? mustSupply="xs:boolean"?>*
            default-property-value?
        </property>

    </constrainingType>
```

The constrainingType element has the following ***attributes***:

- ***name (required)*** – the name of the constrainingType. The form of a constraingType name is an XML QName, in the namespace identified by the targetNamespace attribute.
- ***targetNamespace (optional) –*** an identifier for a target namespace into which the constrainingType is declared
- ***requires (optional)*** – a list of policy intents. See the Policy Framework specification [10] for a description of this attribute.

ConstrainingType contains ***zero or more properties, services***, ***references***.

When an implementation is constrained by a constrainingType it must define all the services, references and properties specified in the corresponding constrainingType. The constraining type's references and services will have interfaces specified and may have intents specified. An implementation may contain additional services, additional optional references and additional optional properties, but cannot contain additional non-optional references or additional non-optional properties (a non-optional property is one with no default value applied).

When a component is constrained by a constrainingType (via the "constrainingType" attribute), the entire componentType associated with the component and its implementation is not visible to the containing composite. The containing composite can only see a projection of the componentType associated with the component and implementation as scoped by the constrainingType of the component. For example, an additional service provided by the implementation which is not in the constrainingType associated with the component cannot be promoted by the containing composite. This requirement ensures that the constrainingType contract cannot be violated by the composite.

The constrainingType can include required intents on any element. Those intents are applied to any component that uses that constrainingType. In other words, if requires="reliability" exists on a constrainingType, or its child service or reference elements, then a constrained component or its implementation must include requires="reliability" on the component or implementation or on its corresponding service or reference. Note that the component or implementation may use a qualified form of an intent specified in unqualified form in the constrainingType, but if the constrainingType uses the qualified form, then the component or implementation must also use the qualified form, otherwise there is an error.

A constrainingType can be applied to an implementation. In this case, the implementation's componentType has a constrainingType attribute set to the QName of the constrainingType.

6.8.1 Example constrainingType

The following snippet shows the contents of the component called "MyValueServiceComponent" which is constrained by the constrainingType myns:CT. The componentType associated with the implementation is also shown.

```
   <component name="MyValueServiceComponent" constrainingType="myns:CT>
     <implementation.java class="services.myvalue.MyValueServiceImpl"/>
     <property name="currency">EURO</property>
     <reference name="customerService" target="CustomerService">
       <binding.ws ...>
     <reference name="StockQuoteService"
         target="StockQuoteMediatorComponent"/>
   </component>

   <constrainingType name="CT"
               targetNamespace="http://myns.com">
     <service name="MyValueService">
       <interface.java interface="services.myvalue.MyValueService"/>
     </service>
     <reference name="customerService">
       <interface.java interface="services.customer.CustomerService"/>
     </reference>
     <reference name="stockQuoteService">
       <interface.java interface="services.stockquote.StockQuoteService"/>
     </reference>
     <property name="currency" type="xsd:string"/>
   </constrainingType>
```

The component MyValueServiceComponent is constrained by the constrainingType CT which means that it must provide:

- service ***MyValueService*** with the interface services.myvalue.MyValueService
- reference ***customerService*** with the interface services.stockquote.StockQuoteService
- reference ***stockQuoteService*** with the interface services.stockquote.StockQuoteService
- property ***currency*** of type xsd:string.

7 Binding

Bindings are used by services and references. References use bindings to describe the access mechanism used to call a service (which can be a service provided by another SCA composite). Services use bindings to describe the access mechanism that clients (which can be a client from another SCA composite) have to use to call the service.

SCA supports the use of multiple different types of bindings. Examples include ***SCA service, Web service, stateless session EJB, data base stored procedure, EIS service***. An SCA runtime MUST provide support for SCA service and Web service binding types. SCA provides an extensibility mechanism by which an SCA runtime can add support for additional binding types. For details on how additional binding types are defined, see the section on the Extension Model.

A binding is defined by a ***binding element*** which is a child element of a service or of a reference element in a composite. The following snippet shows the composite schema with the schema for the binding element.

```
<?xml version="1.0" encoding="ASCII"?>
<!-- Bindings schema snippet -->
<composite     xmlns="http://www.osoa.org/xmlns/sca/1.0"
               targetNamespace="xs:anyURI"
               name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?
               constrainingType="QName"?
               requires="list of xs:QName"? policySets="list of
xs:QName"?>

   ...

   <service name="xs:NCName" promote="xs:anyURI"
         requires="list of xs:QName"? policySets="list of xs:QName"?>*
         <interface/>?
         <binding uri="xs:anyURI"? name="xs:QName"?
               requires="list of xs:QName"? policySets="list of
xs:QName"?/>*
         <callback>?
               <binding uri="xs:anyURI"? name="xs:QName"?
                     requires="list of xs:QName"?
                     policySets="list of xs:QName"?/>+
         </callback>
   </service>

   ...

   <reference name="xs:NCName" target="list of xs:anyURI"?
```

```
            promote="list of xs:anyURI"? wiredByImpl="xs:boolean"?
            multiplicity="0..1 or 1..1 or 0..n or 1..n"?
            requires="list of xs:QName"? policySets="list of xs:QName"?>*
            <interface/>?
            <binding uri="xs:anyURI"? name="xs:QName"?
                  requires="list of xs:QName"? policySets="list of
      xs:QName"?/>*
            <callback>?
                  <binding uri="xs:anyURI"? name="xs:QName"?
                        requires="list of xs:QName"?
                        policySets="list of xs:QName"?/>+
            </callback>
      </reference>

      ...

</composite>
```

The element name of the binding element is architected; it is in itself a qualified name. The first qualifier is always named “binding”, and the second qualifier names the respective binding-type (e.g. binding.composite, binding.ws, binding.ejb, binding.eis).

A binding element has the following attributes:

- ***uri (optional) -*** has the following semantic.
  - o For a binding of a ***reference*** the URI attribute defines the target URI of the reference (either the component/service for a wire to an endpoint within the SCA domain or the accessible address of some endpoint outside the SCA domain). It is optional for references defined in composites used as component implementations, but required for references defined in composites contributed to SCA domains. The URI attribute of a reference of a composite can be reconfigured by a component in a containing composite using the composite as an implementation. Some binding types may require that the address of the target service uses more than a simple URI (such as a WS-Addressing endpoint reference). In those cases, the binding type will define the additional attributes or sub-elements that are necessary to identify the service.
  - o For a binding of a ***service*** the URI attribute defines the URI relative to the component which contributes the service to the SCA domain. The default value for the URI is the the value of the name attribute of the binding.
- ***name (optional)*** – a name for the binding instance (a QName). The name attribute allows distinction between multiple binding elements on a single service or reference. The default value of the name attribute is the service or reference name. When a service or reference has multiple bindings, only one can have the default value; all others must have a value specified that is unique within the service or reference. The name also permits the binding instance to be referenced from elsewhere – particularly useful for some types of binding, which can be declared in a definitions document as a template and referenced from other binding instances, simplifying the definition of more complex binding instances (see the JMS Binding specification [11] for examples of this referencing).
- ***requires (optional)*** - a list of policy intents. See the Policy Framework specification [10] for a description of this attribute.

- ***policySets (optional)*** – a list of policy sets. See the Policy Framework specification [10] for a description of this attribute.

When multiple bindings exist for an service, it means that the service is available by any of the specified bindings. The technique that the SCA runtime uses to choose among available bindings is left to the implementation and it may include additional (nonstandard) configuration. Whatever technique is used SHOULD be documented.

Services and References can always have their bindings overridden at the SCA domain level, unless restricted by Intents applied to them.

The following sections describe the SCA and Web service binding type in detail.

7.1 Messages containing Data not defined in the Service Interface

It is possible for a message to include information that is not defined in the interface used to define the service, for instance information may be contained in SOAP headers or as MIME attachments.

Implementation types MAY make this information available to component implementations in their execution context. These implementation types must indicate how this information is accessed and in what form they are presented.

7.2 Form of the URI of a Deployed Binding

7.2.1 Constructing Hierarchical URIs

Bindings that use hierarchical URI schemes construct the effective URI with a combination of the following pieces:

Base System URI for a scheme / Component URI / Service Binding URI

Each of these components deserves addition definition:

Base Domain URI for a scheme. An SCA domain should define a base URI for each hierarchical URI scheme on which it intends to provide services.

For example: the HTTP and HTTPS schemes would each have their own base URI defined for the domain. An example of a scheme that is not hierarchical, and therefore will have no base URI is the “jms:” scheme.

Component URI. The component URI above is for a component that is deployed in the SCA Domain. The URI of a component defaults to the name of the component, which is used as a relative URI. The component may have a specified URI value. The specified URI value may be an absolute URI in which case it becomes the Base URI for all the services belonging to the component. If the specified URI value is a relative URI, it is used as the Component URI value above.

Service Binding URI. The Service Binding URI is the relative URI specified in the “uri” attribute of a binding element of the service. The default value of the attribute is value of the binding’s name attribute treated as a relative URI. If multiple bindings for a single service use the same scheme (e.g. HTTP), then only one of the bindings may depend on the default value for the uri attribute, i.e. only one may use the default binding name. The service binding URI may also be absolute, in which case the absolute URI fully specifies the full URI of the service. Some deployment environments may not support the use of absolute URIs in service bindings.

Where a component has only a single service, the default value of the Service Binding URI is null, so that the effective URI is:

Base Domain URI for a scheme / Component URI

This shortened form of the URI is consistent with the shortened form for the wire target URI used when wiring references to services

Services deployed into the Domain (as opposed to services of components) have a URI that does not include a component name, i.e.:

Base Domain URI for a scheme / Service Binding URI

The name of the containing composite does not contribute to the URI of any service.

For example, a service where the Base URI is "http://acme.com", the component is named "stocksComponent" and the service binding name is "getQuote", the URI would look like this:

http://acme.com/stocksComponent/getQuote

Allowing a binding's relative URI to be specified that differs from the name of the service allows the URI hierarchy of services to be designed independently of the organization of the domain.

It is good practice to design the URI hierarchy to be independent of the domain organization, but there may be times when domains are initially created using the default URI hierarchy. When this is the case, the organization of the domain can be changed, while maintaining the form of the URI hierarchy, by giving appropriate values to the ***uri*** attribute of select elements. Here is an example of a change that can be made to the organization while maintaining the existing URIs:

To move a subset of the services out of one component (say "foo") to a new component (say "bar"), the new component should have bindings for the moved services specify a URI "../foo/MovedService"..

The URI attribute may also be used in order to create shorter URIs for some endpoints, where the component name may not be present in the URI at all. For example, if a binding has a ***uri*** attribute of "../myService" the component name will not be present in the URI.

7.2.2 Non-hierarchical URIs

Bindings that use non-hierarchical URI schemes (such as jms: or mailto:) may optionally make use of the "uri" attritibute, which is the complete representation of the URI for that service binding. Where the binding does not use the "uri" attribute, the binding must offer a different mechanism for specifying the service address.

7.2.3 Determining the URI scheme of a deployed binding

One of the things that needs to be determined when building the effective URI of a deployed binding (i.e. endpoint) is the URI scheme. The process of determining the endpoint URI scheme is binding type specific.

If the binding type supports a single protocol then there is only one URI scheme associated with it. In this case, that URI scheme is used.

If the binding type supports multiple protocols, the binding type implementation determines the URI scheme by introspecting the binding configuration, which may include the policy sets associated with the binding.

A good example of a binding type that supports multiple protocols is binding.ws, which can be configured by referencing either an "abstract" WSDL element (i.e. portType or interface) or a "concrete" WSDL element (i.e. binding, port or endpoint). When the binding references a PortType or Interface, the protocol and therefore the URI scheme is derived from the intents/policy sets attached to the binding. When the binding references a "concrete" WSDL element, there are two cases:

1) The referenced WSDL binding element uniquely identifies a URI scheme. This is the most common case. In this case, the URI scheme is given by the protocol/transport specified in the WSDL binding element.

2) The referenced WSDL binding element doesn't uniquely identify a URI scheme. For example, when HTTP is specified in the @transport attribute of the SOAP binding element, both "http" and "https" could be used as valid URI schemes. In this case, the URI scheme is determined by looking at the policy sets attached to the binding.

It's worth noting that an intent supported by a binding type may completely change the behavior of the binding. For example, when the intent "confidentiality/transport" is required by an HTTP binding, SSL is turned on. This basically changes the URI scheme of the binding from "http" to "https".

7.3 SCA Binding

The SCA binding element is defined by the following schema.

```
<binding.sca />
```

The SCA binding can be used for service interactions between references and services contained within the SCA domain. The way in which this binding type is implemented is not defined by the SCA specification and it can be implemented in different ways by different SCA runtimes. The only requirement is that the required qualities of service must be implemented for the SCA binding type. The SCA binding type is ***not*** intended to be an interoperable binding type. For interoperability, an interoperable binding type such as the Web service binding should be used.

A service or reference definition with no binding element specified uses the SCA binding. <binding.sca/> would only have to be specified in override cases, or when you specify a set of bindings on a service or reference definition and the SCA binding should be one of them.

If the interface of the service or reference is local, then the local variant of the SCA binding will be used. If the interface of the service or reference is remotable, then either the local or remote variant of the SCA binding will be used depending on whether source and target are co-located or not.

If a reference specifies an URI via its uri attribute, then this provides the default wire to a service provided by another domain level component. The value of the URI has to be as follows:

- <domain-component-name>/<service-name>

7.3.1 Example SCA Binding

The following snippet shows the MyValueComposite.composite file for the MyValueComposite containing the service element for the MyValueService and a reference element for the StockQuoteService. Both the service and the reference use an SCA binding. The target for the reference is left undefined in this binding and would have to be supplied by the composite in which this composite is used.

```
<?xml version="1.0" encoding="ASCII"?>
<!-- Binding SCA example -->
<composite     xmlns="http://www.osoa.org/xmlns/sca/1.0"
               targetNamespace="http://foo.com"
               name="MyValueComposite" >
```

```

      <service name="MyValueService" promote="MyValueComponent">
            <interface.java interface="services.myvalue.MyValueService"/>
            <binding.sca/>
            …
      </service>

      …

      <reference name="StockQuoteService"
   promote="MyValueComponent/StockQuoteReference">
            <interface.java
   interface="services.stockquote.StockQuoteService"/>
            <binding.sca/>
      </reference>

   </composite>
```

7.4 Web Service Binding

SCA defines a Web services binding. This is described in a separate specification document [9].

7.5 JMS Binding

SCA defines a JMS binding. This is described in a separate specification document [11].

8 SCA Definitions

There are a variety of SCA artifacts which are generally useful and which are not specific to a particular composite or a particular component. These shared artifacts include intents, policy sets, bindings, binding type definitions and implementation type definitions.

All of these artifacts within an SCA Domain are defined in a global, SCA Domain-wide file named definitions.xml. The definitions.xml file contains a definitions element that conforms to the following pseudo-schema snippet:

```
<?xml version="1.0" encoding="ASCII"?>
<!-- Composite schema snippet -->
<definitions   xmlns="http://www.osoa.org/xmlns/sca/1.0"
               targetNamespace="xs:anyURI">

   <sca:intent/>*

   <sca:policySet/>*

   <sca:binding/>*

   <sca:bindingType/>*

   <sca:implementationType/>*

</definitions>
```

The definitions element has the following attribute:

- ***targetNamespace (required)*** – the namespace into which the child elements of this definitions element are placed (used for artifact resolution)

The definitions element contains optional child elements – intent, policySet, binding, bindingtype and implementationType. These elements are described elsewhere in this specification or in the SCA Policy Framework specification [10]. The use of the elements declared within a definitions element is described in the SCA Policy Framework specification [10] and in the JMS Binding specification [11].

9 Extension Model

The assembly model can be extended with support for new interface types, implementation types and binding types. The extension model is based on XML schema substitution groups. There are three XML Schema substitution group heads defined in the SCA namespace: ***interface***, ***implementation*** and ***binding***, for interface types, implementation types and binding types, respectively.

The SCA Client and Implementation specifications and the SCA Bindings specifications (see [1]) use these XML Schema substitution groups to define some basic types of interfaces, implementations and bindings, but other types can be defined as required, where support for these extra ones is available from the runtime. The inteface type elements, implementation type elements, and binding type elements defined by the SCA specifications (see [1]) are all part of the SCA namespace ("http://www.osoa.org/xmlns/sca/1.0"), as indicated in their respective schemas. New interface types, implementation types and binding types that are defined using this extensibility model, which are not part of these SCA specifications must be defined in namespaces other than the SCA namespace.

The "." notation is used in naming elements defined by the SCA specifications (e.g. <implementation.java … />, <interface.wsdl … />, <binding.ws … />), not as a parallel extensibility approach but as a naming convention that improves usability of the SCA assembly language.

Note: How to contribute SCA model extensions and their runtime function to an SCA runtime will be defined by a future version of the specification.

9.1 Defining an Interface Type

The following snippet shows the base definition for the ***interface*** element and ***Interface*** type contained in ***sca-core.xsd***; see appendix for complete schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2006 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
        xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
        elementFormDefault="qualified">

   ...

    <element name="interface" type="sca:Interface" abstract="true"/>
    <complexType name="Interface"/>

   ...

</schema>
```

In the following snippet we show how the base definition is extended to support Java interfaces. The snippet shows the definition of the ***interface.java*** element and the ***JavaInterface*** type contained in ***sca-interface-java.xsd***.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema  xmlns="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
         xmlns:sca="http://www.osoa.org/xmlns/sca/1.0">

   <element name="interface.java" type="sca:JavaInterface"
         substitutionGroup="sca:interface"/>
   <complexType name="JavaInterface">
         <complexContent>
               <extension base="sca:Interface">
                     <attribute name="interface" type="NCName"
use="required"/>
               </extension>
         </complexContent>
   </complexType>
</schema>
```

In the following snippet we show an example of how the base definition can be extended by other specifications to support a new interface not defined in the SCA specifications. The snippet shows the definition of the ***my-interface-extension*** element and the ***my-interface-extension-type*** type.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://www.example.org/myextension"
         xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
        xmlns:tns="http://www.example.org/myextension">

   <element name="my-interface-extension" type="tns:my-interface-
extension-type"
         substitutionGroup="sca:interface"/>
   <complexType name="my-interface-extension-type">
         <complexContent>
               <extension base="sca:Interface">
                     ...
               </extension>
         </complexContent>
   </complexType>
</schema>
```

9.2 Defining an Implementation Type

The following snippet shows the base definition for the ***implementation*** element and ***Implementation*** type contained in ***sca-core.xsd***; see appendix for complete schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2006 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
        xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
        elementFormDefault="qualified">

   ...

    <element name="implementation" type="sca:Implementation"
abstract="true"/>
    <complexType name="Implementation"/>

   ...

</schema>
```

In the following snippet we show how the base definition is extended to support Java implementation. The snippet shows the definition of the ***implementation.java*** element and the ***JavaImplementation*** type contained in ***sca-implementation-java.xsd***.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema  xmlns="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
         xmlns:sca="http://www.osoa.org/xmlns/sca/1.0">

   <element name="implementation.java" type="sca:JavaImplementation"
                              substitutionGroup="sca:implementation"/>
   <complexType name="JavaImplementation">
         <complexContent>
               <extension base="sca:Implementation">
                     <attribute name="class" type="NCName"
use="required"/>
               </extension>
         </complexContent>
   </complexType>
</schema>
```

In the following snippet we show an example of how the base definition can be extended by other specifications to support a new implementation type not defined in the SCA specifications. The snippet shows the definition of the ***my-impl-extension*** element and the ***my-impl-extension-type*** type.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://www.example.org/myextension"
         xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
        xmlns:tns="http://www.example.org/myextension">

   <element name="my-impl-extension" type="tns:my-impl-extension-type"
         substitutionGroup="sca:implementation"/>
   <complexType name="my-impl-extension-type">
         <complexContent>
               <extension base="sca:Implementation">
                     ...
               </extension>
         </complexContent>
   </complexType>
</schema>
```

In addition to the definition for the new implementation instance element, there needs to be an associated implementationType element which provides metadata about the new implementation type. The pseudo schema for the implementationType element is shown in the following snippet:

```
<implementationType type="xs:QName"
               alwaysProvides="list of intent xs:QName"
               mayProvide="list of intent xs:QName"/>
```

The implementation type has the following attributes:

- ***type (required)*** – the type of the implementation to which this implementationType element applies. This is intended to be the QName of the implementation element for the implementation type, such as "sca:implementation.java"
- ***alwaysProvides (optional)*** – a set of intents which the implementation type always provides. See the Policy Framework specification [10] for details.
- ***mayProvide (optional)*** – a set of intents which the implementation type may provide. See the Policy Framework specification [10] for details.

9.3 Defining a Binding Type

The following snippet shows the base definition for the ***binding*** element and ***Binding*** type contained in ***sca-core.xsd***; see appendix for complete schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- binding type schema snippet -->
<!-- (c) Copyright SCA Collaboration 2006, 2007 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
        xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
        elementFormDefault="qualified">
```

```

    ...

     <element name="binding" type="sca:Binding" abstract="true"/>
     <complexType name="Binding">
         <attribute name="uri" type="anyURI" use="optional"/>
         <attribute name="name" type="NCName" use="optional"/>
         <attribute name="requires" type="sca:listOfQNames"
use="optional"/>
         <attribute name="policySets" type="sca:listOfQNames"
use="optional"/>
     </complexType>

    ...

</schema>
```

In the following snippet we show how the base definition is extended to support Web service binding. The snippet shows the definition of the ***binding.ws*** element and the ***WebServiceBinding*** type contained in ***sca-binding-webservice.xsd***.

```

<?xml version="1.0" encoding="UTF-8"?>
<schema  xmlns="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
         xmlns:sca="http://www.osoa.org/xmlns/sca/1.0">

   <element name="binding.ws" type="sca:WebServiceBinding"
         substitutionGroup="sca:binding"/>
   <complexType name="WebServiceBinding">
         <complexContent>
               <extension base="sca:Binding">
                     <attribute name="port" type="anyURI" use="required"/>
               </extension>
         </complexContent>
   </complexType>
</schema>
```

In the following snippet we show an example of how the base definition can be extended by other specifications to support a new binding not defined in the SCA specifications. The snippet shows the definition of the ***my-binding-extension*** element and the ***my-binding-extension-type*** type.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://www.example.org/myextension"
         xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
        xmlns:tns="http://www.example.org/myextension">

```

```
   <element name="my-binding-extension" type="tns:my-binding-extension-
type"
         substitutionGroup="sca:binding"/>
   <complexType name="my-binding-extension-type">
         <complexContent>
               <extension base="sca:Binding">
                     ...
               </extension>
         </complexContent>
   </complexType>
</schema>
```

In addition to the definition for the new binding instance element, there needs to be an associated bindingType element which provides metadata about the new binding type. The pseudo schema for the bindingType element is shown in the following snippet:

```
<bindingType type="xs:QName"
          alwaysProvides="list of intent QNames"?
          mayProvide = "list of intent QNames"?/>
```

The binding type has the following attributes:

- ***type (required)*** – the type of the binding to which this bindingType element applies. This is intended to be the QName of the binding element for the binding type, such as "sca:binding.ws"
- ***alwaysProvides (optional)*** – a set of intents which the binding type always provides. See the Policy Framework specification [10] for details.
- ***mayProvide (optional)*** – a set of intents which the binding type may provide. See the Policy Framework specification [10] for details.

10 Packaging and Deployment

10.1 Domains

An ***SCA Domain*** represents a complete runtime configuration, potentially distributed over a series of interconnected runtime nodes.

A single SCA domain defines the boundary of visibility for all SCA mechanisms. For example, SCA wires can only be used to connect components within a single SCA domain. Connections to services outside the domain must use binding specific mechanisms for addressing services (such as WSDL endpoint URIs). Also, SCA mechanisms such as intents and policySets can only be used in the context of a single domain. In general, external clients of a service that is developed and deployed using SCA should not be able to tell that SCA was used to implement the service – it is an implementation detail.

The size and configuration of an SCA Domain is not constrained by the SCA Assembly specification and is expected to be highly variable. An SCA Domain typically represents an area of business functionality controlled by a single organization. For example, an SCA Domain may be the whole of a business, or it may be a department within a business.

As an example, for the accounts department in a business, the SCA Domain might cover all finance-related functions, and it might contain a series of composites dealing with specific areas of accounting, with one for Customer accounts and another dealing with Accounts Payable.

An SCA domain has the following:

- A virtual domain-level composite whose components are deployed and running
- A set of *installed contributions* that contain implementations, interfaces and other artifacts necessary to execute components
- A set of logical services for manipulating the set of contributions and the virtual domain-level composite.

The information associated with an SCA domain can be stored in many ways, including but not limited to a specific filesystem structure or a repository.

10.2 Contributions

An SCA domain may require a large number of different artifacts in order to work. These artifacts include artifacts defined by SCA and other artifacts such as object code files and interface definition files. The SCA-defined artifact types are all XML documents. The root elements of the different SCA definition documents are: composite, componentType, constrainingType and definitions. XML artifacts that are not defined by SCA but which may be needed by an SCA domain include XML Schema documents, WSDL documents, and BPEL documents. SCA constructs, like other XML-defined constructs, use XML qualified names for their identity (i.e. namespace + local name).

Non-XML artifacts are also required within an SCA domain. The most obvious examples of such non-XML artifacts are Java, C++ and other programming language files necessary for component implementations. Since SCA is extensible, other XML and non-XML artifacts may also be required.

SCA defines an interoperable packaging format for contributions (ZIP), as specified below. This format is not the only packaging format that an SCA runtime can use. SCA allows many different packaging formats, but requires that the ZIP format be supported. When using the ZIP format for deploying a contribution, this specification does not specify whether that format is retained after deployment. For example, a Java EE based SCA runtime may convert the ZIP package to an EAR package. SCA expects certain characteristics of any packaging:

- It must be possible to present the artifacts of the packaging to SCA as a hierarchy of resources based off of a single root

- A directory resource should exist at the root of the hierarchy named META-INF
- A document should exist directly under the META-INF directory named sca-contribution.xml which lists the SCA Composites within the contribution that are runnable.

  The same document also optionally lists namespaces of constructs that are defined within the contribution and which may be used by other contributions
  Optionally, additional elements may exist that list the namespaces of constructs that are needed by the contribution and which must be found elsewhere, for example in other contributions. These optional elements may not be physically present in the packaging, but may be generated based on the definitions and references that are present, or they may not exist at all if there are no unresolved references.

  See the section "SCA Contribution Metadata Document" for details of the format of this file.

To illustrate that a variety of packaging formats can be used with SCA, the following are examples of formats that might be used to package SCA artifacts and metadata (as well as other artifacts) as a contribution:

- A filesystem directory
- An OSGi bundle
- A compressed directory (zip, gzip, etc)
- A JAR file (or its variants – WAR, EAR, etc)

Contributions do not contain other contributions. If the packaging format is a JAR file that contains other JAR files (or any similar nesting of other technologies), the internal files are not treated as separate SCA contributions. It is up to the implementation to determine whether the internal JAR file should be represented as a single artifact in the contribution hierarchy or whether all of the contents should be represented as separate artifacts.

A goal of SCA's approach to deployment is that the contents of a contribution should not need to be modified in order to install and use the contents of the contribution in a domain.

10.2.1 SCA Artifact Resolution

Contributions may be self-contained, in that all of the artifacts necessary to run the contents of the contribution are found within the contribution itself. However, it may also be the case that the contents of the contribution make one or many references to artifacts that are not contained within the contribution. These references may be to SCA artifacts or they may be to other artifacts such as WSDL files, XSD files or to code artifacts such as Java class files and BPEL scripts.

A contribution may use some artifact-related or packaging-related means to resolve artifact references. Examples of such mechanisms include:

- wsdlLocation and schemaLocation attributes in references to WSDL and XSD schema artifacts respectively
- OSGi bundle mechanisms for resolving Java class and related resource dependencies

Where present, these mechanisms must be used to resolve artifact dependencies.

SCA also provides an artifact resolution mechanism. The SCA artifact resolution mechanisms are used either where no other mechanisms are available, or in cases where the mechanisms used by the various contributions in the same SCA Domain are different. An example of the latter case is where an OSGi Bundle is used for one contribution but where a second contribution used by the first one is not implemented using OSGi - eg the second contribution is a mainframe COBOL service whose interfaces are declared using WSDL which must be accessed by the first contribution.

The SCA artifact resolution is likely to be most useful for SCA domains containing heterogeneous mixtures of contribution, where artifact-related or packaging-related mechanisms are unlikely to work across different kinds of contribution.

SCA artifact resolution works on the principle that a contribution which needs to use artifacts defined elsewhere expresses these dependencies using ***import*** statements in metadata belonging to the contribution. A contribution controls which artifacts it makes available to other contributions through ***export*** statements in metadata attached to the contribution.

10.2.2 SCA Contribution Metadata Document

The contribution optionally contains a document that declares runnable composites, exported definitions and imported definitions. The document is found at the path of META-INF/sca-contribution.xml relative to the root of the contribution. Frequently some SCA metadata may need to be specified by hand while other metadata is generated by tools (such as the <import> elements described below). To accommodate this, it is also possible to have an identically structured document at META-INF/sca-contribution-generated.xml. If this document exists (or is generated on an as-needed basis), it will be merged into the contents of sca-contribution.xml, with the entries in sca-contribution.xml taking priority if there are any conflicting declarations.

The format of the document is:

```
<?xml version="1.0" encoding="ASCII"?>
<!-- sca-contribution pseudo-schema -->
<contribution xmlns=http://www.osoa.org/xmlns/sca/1.0>

   <deployable composite="xs:QName"/>*
   <import namespace="xs:String" location=”xs:AnyURI”?/>*
   <export namespace="xs:String"/>*

</contribution>
```

deployable element: Identifies a composite which is a composite within the contribution that is a composite intended for potential inclusion into the virtual domain-level composite. Other composites in the contribution are not intended for inclusion but only for use by other composites. New composites can be created for a contribution after it is installed, by using the add Deployment Composite capability and the add To Domain Level Composite capability.

- ***composite (required)*** – The QName of a composite within the contribution.

Export element: A declaration that artifacts belonging to a particular namespace are exported and are available for use within other contributions. An export declaration in a contribution specifies a namespace, all of whose definitions are considered to be exported. By default, definitions are not exported.

The SCA artifact export is useful for SCA domains containing heterogeneous mixtures of contribution packagings and technologies, where artifact-related or packaging-related mechanisms are unlikely to work across different kinds of contribution.

- ***namespace (required)*** – For XML definitions, which are identified by QNames, the namespace should be the namespace URI for the exported definitions. For XML technologies that define multiple *symbol spaces* that can be used within one namespace (e.g. WSDL port types are a different symbol space from WSDL bindings), all definitions from all symbol spaces are exported.

  Technologies that use naming schemes other than QNames must use a different export element from the same substitution group as the the SCA <export> element. The element used identifies the technology, and may use any value for the namespace that is appropriate for that technology. For example, <export.java> can be used can be used to export java definitions, in which case the namespace should be a fully qualified package name.

Import element: Import declarations specify namespaces of definitions that are needed by the definitions and implementations within the contribution, but which are not present in the contribution. It is expected that in most cases import declarations will be generated based on introspection of the contents of the contribution. In this case, the import declarations would be found in the META-INF/ sca-contribution-generated.xml document.

- ***namespace (required)*** – For XML definitions, which are identified by QNames, the namespace should be the namespace URI for the imported definitions. For XML technologies that define multiple *symbol spaces* that can be used within one namespace (e.g. WSDL port types are a different symbol space from WSDL bindings), all definitions from all symbol spaces are imported.

  Technologies that use naming schemes other than QNames must use a different import element from the same substitution group as the the SCA <import> element. The element used identifies the technology, and may use any value for the namespace that is appropriate for that technology. For example, <import.java> can be used can be used to import java definitions, in which case the namespace should be a fully qualified package name.
- ***location (optional)*** – a URI to resolve the definitions for this import. SCA makes no specific requirements for the form of this URI, nor the means by which it is resolved. It may point to another contribution (through its URI) or it may point to some location entirely outside the SCA Domain.

It is expected that SCA runtimes may define implementation specific ways of resolving location information for artifact resolution between contributions. These mechanisms will however usually be limited to sets of contributions of one runtime technology and one hosting environment.

In order to accommodate imports of artifacts between contributions of disparate runtime technologies, it is strongly suggested that SCA runtimes honor SCA contribution URIs as location specification.

SCA runtimes that support contribution URIs for cross-contribution resolution of SCA artifacts should do so similarly when used as @schemaLocation and @wsdlLocation and other artifact location specifications.

The order in which the import statements are specified may play a role in this mechanism. Since definitions of one namespace can be distributed across several artifacts, multiple import declarations can be made for one namespace.

The location value is only a default, and dependent contributions listed in the call to installContribution should override the value if there is a conflict. However, the specific mechanism for resolving conflicts between contributions that define conflicting definitions is implementation specific.

If the value of the location attribute is an SCA contribution URI, then the contribution packaging may become dependent on the deployment environment. In order to avoid such a dependency, dependent contributions should be specified only when deploying or updating contributions as specified in the section 'Operations for Contributions' below.

10.2.3 Contribution Packaging using ZIP

SCA allows many different packaging formats that SCA runtimes can support, but SCA requires that all runtimes support the ZIP packaging format for contributions. This format allows that metadata specified by the section 'SCA Contribution Metadata Document' be present. Specifically, it may contain a top-level "META-INF" directory and a "META-INF/sca-contribution.xml" file and there may also be an optional "META-INF/sca-contribution-generated.xml" file in the package. SCA defined artifacts as well as non-SCA defined artifacts such as object files, WSDL definition, Java classes may be present anywhere in the ZIP archive,

A up to date definition of the ZIP file format is published by PKWARE in an Application Note on the .ZIP file format [12].

10.3 Installed Contribution

As noted in the section above, the contents of a contribution should not need to be modified in order to install and use it within a domain. An *installed contribution* is a contribution with all of the associated information necessary in order to execute *deployable composites* within the contribution.

An installed contribution is made up of the following things:

- Contribution Packaging – the contribution that will be used as the starting point for resolving all references
- Contribution base URI
- Dependent contributions: a set of snapshots of other contributions that are used to resolve the import statements from the root composite and from other dependent contributions
    - Dependent contributions may or may not be shared with other installed contributions.
    - When the snapshot of any contribution is taken is implementation defined, ranging from the time the contribution is installed to the time of execution
- Deployment-time composites.
  These are composites that are added into an installed contribution after it has been deployed. This makes it possible to provide final configuration and access to implementations within a contribution without having to modify the contribution. These are optional, as composites that already exist within the contribution may also be used for deployment.

Installed contributions provide a context in which to resolve qualified names (e.g. QNames in XML, fully qualified class names in Java).

If multiple dependent contributions have exported definitions with conflicting qualified names, the algorithm used to determine the qualified name to use is implementation dependent. Implementations of SCA may also generate an error if there are conflicting names.

10.3.1 Installed Artifact URIs

When a contribution is installed, all artifacts within the contribution are assigned URIs, which are constructed by starting with the base URI of the contribution and adding the relative URI of each artifact (recalling that SCA requires that any packaging format be able to offer up its artifacts in a single hierarchy).

10.4 Operations for Contributions

SCA Domains provide the following conceptual functionality associated with contributions (meaning the function may not be represented as addressable services and also meaning that equivalent functionality may be provided in other ways). The functionality is optional meaning that some SCA runtimes may choose not to provide that functionality in any way:

10.4.1 install Contribution & update Contribution

Creates or updates an installed contribution with a supplied root contribution, and installed at a supplied base URI. A supplied dependent contribution list specifies the contributions that should be used to resolve the dependencies of the root contribution and other dependent contributions. These override any dependent contributions explicitly listed via the location attribute in the import statements of the contribution.

SCA follows the simplifying assumption that the use of a contribution for resolving anything also means that all other exported artifacts can be used from that contribution. Because of this, the dependent contribution list is just a list of installed contribution URIs. There is no need to specify what is being used from each one.

Each dependent contribution is also an installed contribution, with its own dependent contributions. By default these dependent contributions of the dependent contributions (which we will call *indirect dependent contributions*) are included as dependent contributions of the installed contribution. However, if a contribution in the dependent contribution list exports any conflicting definitions with an indirect dependent contribution, then the indirect dependent contribution is not included (i.e. the explicit list overrides the default inclusion of indirect dependent contributions). Also, if there is ever a conflict between two indirect dependent contributions, then the conflict must be resolved by an explicit entry in the dependent contribution list.

Note that in many cases, the dependent contribution list can be generated. In particular, if a domain is careful to avoid creating duplicate definitions for the same qualified name, then it is easy for this list to be generated by tooling.

10.4.2 add Deployment Composite & update Deployment Composite

Adds or updates a deployment composite using a supplied composite ("composite by value" – a data structure, not an existing resource in the domain) to the contribution identified by a supplied contribution URI. The added or updated deployment composite is given a relative URI that matches the @name attribute of the composite, with a ".composite" suffix. Since all composites must run within the context of a installed contribution (any component implementations or other definitions are resolved within that contribution), this functionality makes it possible for the deployer to create a composite with final configuration and wiring decisions and add it to an installed contribution without having to modify the contents of the root contribution.

Also, in some use cases, a contribution may include only implementation code (e.g. PHP scripts). It should then be possible for those to be given component names by a (possibly generated) composite that is added into the installed contribution, without having to modify the packaging.

10.4.3 remove Contribution

Removes the deployed contribution identified by a supplied contribution URI.

10.5 Use of Existing (non-SCA) Mechanisms for Resolving Artifacts

For certain types of artifact, there are existing and commonly used mechanisms for referencing a specific concrete location where the artifact can be resolved.

Examples of these mechanisms include:

- For WSDL files, the ***@wsdlLocation*** attribute is a hint that has a URI value pointing to the place holding the WSDL itself.

- For XSDs, the ***@schemaLocation*** attribute is a hint which matches the namespace to a URI where the XSD is found.

Note: In neither of these cases is the runtime obliged to use the location hint and the URI does not have to be dereferenced.

SCA permits the use of these mechanisms. Where present, these mechanisms take precendence over the SCA mechanisms. However, use of these mechanisms is discouraged because tying assemblies to addresses in this way makes the assemblies less flexible and prone to errors when changes are made to the overall SCA Domain.

Note: If one of these mechanisms is present, but there is a failure to find the resource indicated when using the mechanism (eg the URI is incorrect or invalid, say) the SCA runtime MUST raise an error and MUST NOT attempt to use SCA resolution mechanisms as an alternative.

10.6 Domain-Level Composite

The domain-level composite is a virtual composite, in that it is not defined by a composite definition document. Rather, it is built up and modified through operations on the domain. However, in other respects it is very much like a composite, since it contains components, wires, services and references.

The abstract domain-level functionality for modifying the domain-level composite is as follows, although a runtime may supply equivalent functionality in a different form:

10.6.1 add To Domain-Level Composite

This functionality adds the composite identified by a supplied URI to the Domain Level Composite. The supplied composite URI must refer to a composite within a installed contribution. The composite's installed contribution determines how the composite's artifacts are resolved (directly and indirectly). The supplied composite is added to the domain composite with semantics that correspond to the domain-level composite having an <include> statement that references the supplied composite. All of the composite's components become *top-level* components and the services become externally visible services (eg. they would be present in a WSDL description of the domain).

10.6.2 remove From Domain-Level Composite

Removes from the Domain Level composite the elements corresponding to the composite identified by a supplied composite URI. This means that the removal of the components, wires, services and references originally added to the domain level composite by the identified composite.

10.6.3 get Domain-Level Composite

Returns a <composite> definition that has an <include> line for each composite that had been added to the domain level composite. It is important to note that, in dereferencing the included composites, any referenced artifacts must be resolved in terms of that installed composite.

10.6.4 get QName Definition

In order to make sense of the domain-level composite (as returned by get Domain-Level Composite), it must be possible to get the definitions for named artifacts in the included composites. This functionality takes the supplied URI of an installed contribution (which provides the context), a supplied qualified name of a definition to look up, and a supplied symbol space (as a QName, eg wsdl:PortType). The result is a single definition, in whatever form is appropriate for that definition type.

Note that this, like all the other domain-level operations, is a conceptual operation. Its capabilities should exist in some form, but not necessarily as a service operation with exactly this signature.

A. XML Schemas

A.1 sca.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2006 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
    xmlns:sca="http://www.osoa.org/xmlns/sca/1.0">

    <include schemaLocation="sca-core.xsd"/>

    <include schemaLocation="sca-interface-java.xsd"/>
    <include schemaLocation="sca-interface-wsdl.xsd"/>

    <include schemaLocation="sca-implementation-java.xsd"/>
    <include schemaLocation="sca-implementation-composite.xsd"/>

    <include schemaLocation="sca-binding-webservice.xsd"/>
    <include schemaLocation="sca-binding-jms.xsd"/>
    <include schemaLocation="sca-binding-sca.xsd"/>

    <include schemaLocation="sca-definitions.xsd"/>
    <include schemaLocation="sca-policy.xsd"/>

</schema>
```

A.2 sca-core.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2006, 2007 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
        xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
        elementFormDefault="qualified">

    <element name="componentType" type="sca:ComponentType"/>
    <complexType name="ComponentType">
      <sequence>
            <element ref="sca:implementation" minOccurs="0" maxOccurs="1"/>
            <choice minOccurs="0" maxOccurs="unbounded">
                  <element name="service" type="sca:ComponentService" />
                  <element name="reference" type="sca:ComponentReference"/>
                  <element name="property" type="sca:Property"/>
            </choice>
            <any namespace="##other" processContents="lax" minOccurs="0"
                  maxOccurs="unbounded"/>
        </sequence>
        <attribute name="constrainingType" type="QName" use="optional"/>
        <anyAttribute namespace="##any" processContents="lax"/>
    </complexType>

    <element name="composite" type="sca:Composite"/>
    <complexType name="Composite">
         <sequence>
             <element name="include" type="anyURI" minOccurs="0"
                  maxOccurs="unbounded"/>
             <choice minOccurs="0" maxOccurs="unbounded">
                 <element name="service" type="sca:Service"/>
                 <element name="property" type="sca:Property"/>
                 <element name="component" type="sca:Component"/>
                 <element name="reference" type="sca:Reference"/>
                 <element name="wire" type="sca:Wire"/>
             </choice>
             <any namespace="##other" processContents="lax" minOccurs="0"
                  maxOccurs="unbounded"/>
        </sequence>
        <attribute name="name" type="NCName" use="required"/>
        <attribute name="targetNamespace" type="anyURI" use="required"/>
        <attribute name="local" type="boolean" use="optional"
default="false"/>
        <attribute name="autowire" type="boolean" use="optional"
default="false"/>
        <attribute name="constrainingType" type="QName" use="optional"/>
        <attribute name="requires" type="sca:listOfQNames" use="optional"/>
        <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
        <anyAttribute namespace="##any" processContents="lax"/>
    </complexType>

    <complexType name="Service">
      <sequence>
            <element ref="sca:interface" minOccurs="0" maxOccurs="1" />
            <element name="operation" type="sca:Operation" minOccurs="0"
```

```
                  maxOccurs="unbounded" />
            <choice minOccurs="0" maxOccurs="unbounded">
                  <element ref="sca:binding" />
                  <any namespace="##other" processContents="lax"
                        minOccurs="0" maxOccurs="unbounded" />
            </choice>
            <element ref="sca:callback" minOccurs="0" maxOccurs="1" />
            <any namespace="##other" processContents="lax" minOccurs="0"
                  maxOccurs="unbounded" />
      </sequence>
      <attribute name="name" type="NCName" use="required" />
      <attribute name="promote" type="anyURI" use="required" />
      <attribute name="requires" type="sca:listOfQNames" use="optional" />
      <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
      <anyAttribute namespace="##any" processContents="lax" />
    </complexType>

    <element name="interface" type="sca:Interface" abstract="true" />
    <complexType name="Interface" abstract="true"/>

    <complexType name="Reference">
      <sequence>
            <element ref="sca:interface" minOccurs="0" maxOccurs="1" />
            <element name="operation" type="sca:Operation" minOccurs="0"
                  maxOccurs="unbounded" />
            <choice minOccurs="0" maxOccurs="unbounded">
                  <element ref="sca:binding" />
                  <any namespace="##other" processContents="lax" />
            </choice>
            <element ref="sca:callback" minOccurs="0" maxOccurs="1" />
            <any namespace="##other" processContents="lax" minOccurs="0"
                  maxOccurs="unbounded" />
      </sequence>
      <attribute name="name" type="NCName" use="required" />
      <attribute name="target" type="sca:listOfAnyURIs" use="optional"/>
      <attribute name="wiredByImpl" type="boolean" use="optional"
default="false"/>
      <attribute name="multiplicity" type="sca:Multiplicity"
            use="optional" default="1..1" />
      <attribute name="promote" type="sca:listOfAnyURIs" use="required" />
      <attribute name="requires" type="sca:listOfQNames" use="optional" />
      <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
      <anyAttribute namespace="##any" processContents="lax" />
    </complexType>

    <complexType name="SCAPropertyBase" mixed="true">
      <!-- mixed="true" to handle simple type -->
      <sequence>
            <any namespace="##any" processContents="lax" minOccurs="0"
                  maxOccurs="1" />
            <!-- NOT an extension point; This xsd:any exists to accept
                   the element-based or complex type property
                   i.e. no element-based extension point under "sca:property"
-->
      </sequence>
    </complexType>

    <!-- complex type for sca:property declaration -->
    <complexType name="Property" mixed="true">
       <complexContent>
             <extension base="sca:SCAPropertyBase">
                <!-- extension defines the place to hold default value -->
                <attribute name="name" type="NCName" use="required"/>
                <attribute name="type" type="QName" use="optional"/>
                <attribute name="element" type="QName" use="optional"/>
                <attribute name="many" type="boolean" default="false"
                  use="optional"/>
                <attribute name="mustSupply" type="boolean" default="false"
                  use="optional"/>
                <anyAttribute namespace="##any" processContents="lax"/>
                <!-- an extension point ; attribute-based only -->
             </extension>
       </complexContent>
    </complexType>

    <complexType name="PropertyValue" mixed="true">
       <complexContent>
          <extension base="sca:SCAPropertyBase">
             <attribute name="name" type="NCName" use="required"/>
             <attribute name="type" type="QName" use="optional"/>
             <attribute name="element" type="QName" use="optional"/>
             <attribute name="many" type="boolean" default="false"
                  use="optional"/>
             <attribute name="source" type="string" use="optional"/>
             <attribute name="file" type="anyURI" use="optional"/>
             <anyAttribute namespace="##any" processContents="lax"/>
             <!-- an extension point ; attribute-based only -->
          </extension>
       </complexContent>
    </complexType>

    <element name="binding" type="sca:Binding" abstract="true"/>
    <complexType name="Binding" abstract="true">
      <sequence>
            <element name="operation" type="sca:Operation" minOccurs="0"
                  maxOccurs="unbounded" />
      </sequence>
        <attribute name="uri" type="anyURI" use="optional"/>
        <attribute name="name" type="NCName" use="optional"/>
        <attribute name="requires" type="sca:listOfQNames" use="optional"/>
        <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
    </complexType>

    <element name="bindingType" type="sca:BindingType"/>
    <complexType name="BindingType">
      <sequence minOccurs="0" maxOccurs="unbounded">
            <any namespace="##other" processContents="lax" />
      </sequence>
      <attribute name="type" type="QName" use="required"/>
      <attribute name="alwaysProvides" type="sca:listOfQNames"
use="optional"/>
      <attribute name="mayProvide" type="sca:listOfQNames" use="optional"/>
      <anyAttribute namespace="##any" processContents="lax"/>
    </complexType>

    <element name="callback" type="sca:Callback"/>
    <complexType name="Callback">
      <choice minOccurs="0" maxOccurs="unbounded">
            <element ref="sca:binding"/>
            <any namespace="##other" processContents="lax"/>
        </choice>
        <attribute name="requires" type="sca:listOfQNames" use="optional"/>
        <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
        <anyAttribute namespace="##any" processContents="lax"/>
    </complexType>

    <complexType name="Component">
        <sequence>
             <element ref="sca:implementation" minOccurs="0" maxOccurs="1"/>
             <choice minOccurs="0" maxOccurs="unbounded">
                  <element name="service" type="sca:ComponentService"/>
                <element name="reference" type="sca:ComponentReference"/>
                <element name="property" type="sca:PropertyValue" />
             </choice>
             <any namespace="##other" processContents="lax" minOccurs="0"
                  maxOccurs="unbounded"/>
        </sequence>
        <attribute name="name" type="NCName" use="required"/>
        <attribute name="autowire" type="boolean" use="optional"
default="false"/>
        <attribute name="constrainingType" type="QName" use="optional"/>
        <attribute name="requires" type="sca:listOfQNames" use="optional"/>
        <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
        <anyAttribute namespace="##any" processContents="lax"/>
    </complexType>

    <complexType name="ComponentService">
      <complexContent>
            <restriction base="sca:Service">
                    <sequence>
                         <element ref="sca:interface" minOccurs="0"
maxOccurs="1"/>
                         <element name="operation" type="sca:Operation"
minOccurs="0"
                              maxOccurs="unbounded" />
                         <choice minOccurs="0" maxOccurs="unbounded">
                            <element ref="sca:binding"/>
                              <any namespace="##other" processContents="lax"
                                    minOccurs="0" maxOccurs="unbounded"/>
                         </choice>
                         <element ref="sca:callback" minOccurs="0"
maxOccurs="1"/>
                         <any namespace="##other" processContents="lax"
minOccurs="0"
                              maxOccurs="unbounded"/>
                    </sequence>
                    <attribute name="name" type="NCName" use="required"/>
                    <attribute name="requires" type="sca:listOfQNames"
                        use="optional"/>
                    <attribute name="policySets" type="sca:listOfQNames"
                        use="optional"/>
                    <anyAttribute namespace="##any" processContents="lax"/>
            </restriction>
      </complexContent>
    </complexType>

    <complexType name="ComponentReference">
      <complexContent>
            <restriction base="sca:Reference">
                  <sequence>
                        <element ref="sca:interface" minOccurs="0"
maxOccurs="1" />
                        <element name="operation" type="sca:Operation"
minOccurs="0"
                              maxOccurs="unbounded" />
                        <choice minOccurs="0" maxOccurs="unbounded">
                              <element ref="sca:binding" />
                              <any namespace="##other" processContents="lax"
/>
                        </choice>
                        <element ref="sca:callback" minOccurs="0"
maxOccurs="1" />
                        <any namespace="##other" processContents="lax"
minOccurs="0"
                              maxOccurs="unbounded" />
                  </sequence>
                  <attribute name="name" type="NCName" use="required" />
                  <attribute name="autowire" type="boolean" use="optional"
                        default="false"/>
                  <attribute name="wiredByImpl" type="boolean" use="optional"
                        default="false"/>
                  <attribute name="target" type="sca:listOfAnyURIs"
use="optional"/>
                  <attribute name="multiplicity" type="sca:Multiplicity"
                        use="optional" default="1..1" />
                  <attribute name="requires" type="sca:listOfQNames"
use="optional"/>
                  <attribute name="policySets" type="sca:listOfQNames"
                        use="optional"/>
                  <anyAttribute namespace="##any" processContents="lax" />
            </restriction>
      </complexContent>
    </complexType>

    <element name="implementation" type="sca:Implementation"
      abstract="true" />
    <complexType name="Implementation" abstract="true">
      <attribute name="requires" type="sca:listOfQNames" use="optional"/>
```

```
        <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
      </complexType>

      <element name="implementationType" type="sca:ImplementationType"/>
      <complexType name="ImplementationType">
        <sequence minOccurs="0" maxOccurs="unbounded">
              <any namespace="##other" processContents="lax" />
        </sequence>
        <attribute name="type" type="QName" use="required"/>
        <attribute name="alwaysProvides" type="sca:listOfQNames"
use="optional"/>
        <attribute name="mayProvide" type="sca:listOfQNames" use="optional"/>
        <anyAttribute namespace="##any" processContents="lax"/>
      </complexType>

      <complexType name="Wire">
          <sequence>
              <any namespace="##other" processContents="lax" minOccurs="0"
                    maxOccurs="unbounded"/>
          </sequence>
          <attribute name="source" type="anyURI" use="required"/>
          <attribute name="target" type="anyURI" use="required"/>
          <anyAttribute namespace="##any" processContents="lax"/>
      </complexType>

      <element name="include" type="sca:Include"/>
      <complexType name="Include">
              <attribute name="name" type="QName"/>
              <anyAttribute namespace="##any" processContents="lax"/>
        </complexType>

      <complexType name="Operation">
        <attribute name="name" type="NCName" use="required"/>
        <attribute name="requires" type="sca:listOfQNames" use="optional"/>
        <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
        <anyAttribute namespace="##any" processContents="lax"/>
      </complexType>

      <element name="constrainingType" type="sca:ConstrainingType"/>
      <complexType name="ConstrainingType">
          <sequence>
              <choice minOccurs="0" maxOccurs="unbounded">
                  <element name="service" type="sca:ComponentService"/>
```

```
                <element name="reference" type="sca:ComponentReference"/>
                <element name="property" type="sca:Property" />
            </choice>
            <any namespace="##other" processContents="lax" minOccurs="0"
                  maxOccurs="unbounded"/>
        </sequence>
        <attribute name="name" type="NCName" use="required"/>
        <attribute name="targetNamespace" type="anyURI"/>
        <attribute name="requires" type="sca:listOfQNames" use="optional"/>
        <anyAttribute namespace="##any" processContents="lax"/>
    </complexType>


    <simpleType name="Multiplicity">
        <restriction base="string">
            <enumeration value="0..1"/>
            <enumeration value="1..1"/>
            <enumeration value="0..n"/>
            <enumeration value="1..n"/>
        </restriction>
    </simpleType>

    <simpleType name="OverrideOptions">
        <restriction base="string">
            <enumeration value="no"/>
            <enumeration value="may"/>
            <enumeration value="must"/>
        </restriction>
    </simpleType>

    <!-- Global attribute definition for @requires to permit use of intents
         within WSDL documents -->
    <attribute name="requires" type="sca:listOfQNames"/>

    <!-- Global attribute defintion for @endsConversation to mark operations
         as ending a conversation -->
    <attribute name="endsConversation" type="boolean" default="false"/>

    <simpleType name="listOfQNames">
      <list itemType="QName"/>
    </simpleType>

    <simpleType name="listOfAnyURIs">
```

```
        <list itemType="anyURI"/>
    </simpleType>

</schema>
```

A.3 sca-binding-sca.xsd

```

<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2006, 2007 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
    xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
        elementFormDefault="qualified">

    <include schemaLocation="sca-core.xsd"/>

    <element name="binding.sca" type="sca:SCABinding"
      substitutionGroup="sca:binding"/>
    <complexType name="SCABinding">
        <complexContent>
            <extension base="sca:Binding">
                  <sequence>
                        <element name="operation" type="sca:Operation"
minOccurs="0"
                              maxOccurs="unbounded" />
                  </sequence>
                    <attribute name="uri" type="anyURI" use="optional"/>
                    <attribute name="name" type="QName" use="optional"/>
                    <attribute name="requires" type="sca:listOfQNames"
                        use="optional"/>
                    <attribute name="policySets" type="sca:listOfQNames"
                        use="optional"/>
                <anyAttribute namespace="##any" processContents="lax"/>
            </extension>
        </complexContent>
    </complexType>
</schema>

```

A.4 sca-interface-java.xsd

```

<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2006 -->
```

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
    xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
        elementFormDefault="qualified">

    <include schemaLocation="sca-core.xsd"/>

    <element name="interface.java" type="sca:JavaInterface"
                  substitutionGroup="sca:interface"/>
    <complexType name="JavaInterface">
        <complexContent>
            <extension base="sca:Interface">
                <sequence>
                    <any namespace="##other" processContents="lax"
minOccurs="0"                                   maxOccurs="unbounded"/>
                </sequence>
                <attribute name="interface" type="NCName" use="required"/>
                <attribute name="callbackInterface" type="NCName"
use="optional"/>
                <anyAttribute namespace="##any" processContents="lax"/>
            </extension>
        </complexContent>
    </complexType>
</schema>
```

A.5 sca-interface-wsdl.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2006 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
    xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
        elementFormDefault="qualified">

    <include schemaLocation="sca-core.xsd"/>

    <element name="interface.wsdl" type="sca:WSDLPortType"
                  substitutionGroup="sca:interface"/>
    <complexType name="WSDLPortType">
        <complexContent>
            <extension base="sca:Interface">
                <sequence>
                    <any namespace="##other" processContents="lax"
minOccurs="0"                                   maxOccurs="unbounded"/>
```

```
                </sequence>
                <attribute name="interface" type="anyURI" use="required"/>
                <attribute name="callbackInterface" type="anyURI"
use="optional"/>
                <anyAttribute namespace="##any" processContents="lax"/>
            </extension>
        </complexContent>
    </complexType>
</schema>

```

A.6 sca-implementation-java.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2006 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
    xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
    elementFormDefault="qualified">

    <include schemaLocation="sca-core.xsd"/>

    <element name="implementation.java" type="sca:JavaImplementation"
      substitutionGroup="sca:implementation"/>
    <complexType name="JavaImplementation">
        <complexContent>
            <extension base="sca:Implementation">
                <sequence>
                    <any namespace="##other" processContents="lax"
                        minOccurs="0" maxOccurs="unbounded"/>
                </sequence>
                <attribute name="class" type="NCName" use="required"/>
                <attribute name="requires" type="sca:listOfQNames"
use="optional"/>
                  <attribute name="policySets" type="sca:listOfQNames"
                        use="optional"/>
                <anyAttribute namespace="##any" processContents="lax"/>
            </extension>
        </complexContent>
    </complexType>
</schema>
```

A.7 sca-implementation-composite.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2006 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
    xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
    elementFormDefault="qualified">

    <include schemaLocation="sca-core.xsd"/>
    <element name="implementation.composite" type="sca:SCAImplementation"
        substitutionGroup="sca:implementation"/>
    <complexType name="SCAImplementation">
        <complexContent>
            <extension base="sca:Implementation">
                <sequence>
                    <any namespace="##other" processContents="lax"
minOccurs="0"
                        maxOccurs="unbounded"/>
                </sequence>
                <attribute name="name" type="QName" use="required"/>
                <attribute name="requires" type="sca:listOfQNames"
use="optional"/>
                  <attribute name="policySets" type="sca:listOfQNames"
                        use="optional"/>
                <anyAttribute namespace="##any" processContents="lax"/>
            </extension>
        </complexContent>
    </complexType>
</schema>
```

A.8 sca-definitions.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2006 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
    xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
    elementFormDefault="qualified">

    <include schemaLocation="sca-core.xsd"/>

    <element name="definitions">
       <complexType>
          <choice minOccurs="0" maxOccurs="unbounded">
             <element ref="sca:intent"/>
             <element ref="sca:policySet"/>
             <element ref="sca:binding"/>
             <element ref="sca:bindingType"/>
             <element ref="sca:implementationType"/>
             <any namespace="##other" processContents="lax" minOccurs="0"
                  maxOccurs="unbounded"/>
          </choice>
       </complexType>
    </element>

</schema>
```

A.9 sca-binding-webservice.xsd

Is described in the SCA Web Services Binding specification [9]

A.10 sca-binding-jms.xsd

Is described in the SCA JMS Binding specification [11]

A.11 sca-policy.xsd

Is described in the SCA Policy Framework specification [10]

B. SCA Concepts

B.1 Binding

Bindings are used by services and references. References use bindings to describe the access mechanism used to call the service to which they are wired. Services use bindings to describe the access mechanism(s) that clients should use to call the service.

SCA supports multiple different types of bindings. Examples include ***SCA service, Web service, stateless session EJB, data base stored procedure, EIS service.*** SCA provides an extensibility mechanism by which an SCA runtime can add support for additional binding types.

B.2 Component

SCA components are configured instances of ***SCA implementations***, which provide and consume services. SCA allows many different implementation technologies such as Java, BPEL, C++. SCA defines an ***extensibility mechanism*** that allows you to introduce new implementation types. The current specification does not mandate the implementation technologies to be supported by an SCA run-time, vendors may choose to support the ones that are important for them. A single SCA implementation may be used by multiple Components, each with a different configuration.

The Component has a reference to an implementation of which it is an instance, a set of property values, and a set of service reference values. Property values define the values of the properties of the component as defined by the component's implementation. Reference values define the services that resolve the references of the component as defined by its implementation. These values can either be a particular service of a particular component, or a reference of the containing composite.

B.3 Service

SCA services are used to declare the externally accessible services of an ***implementation***. For a composite, a service is typically provided by a service of a component within the composite, or by a reference defined by the composite. The latter case allows the republication of a service with a new address and/or new bindings. The service can be thought of as a point at which messages from external clients enter a composite or implementation.

A service represents an addressable set of operations of an implementation that are designed to be exposed for use by other implementations or exposed publicly for use elsewhere (eg public Web services for use by other organizations). The operations provided by a service are specified by an Interface, as are the operations required by the service client (if there is one). An implementation may contain multiple services, when it is possible to address the services of the implementation separately.

A service may be provided ***as SCA remote services, as Web services, as stateless session EJB's, as EIS services, and so on***. Services use ***bindings*** to describe the way in which they are published. SCA provides an ***extensibility mechanism*** that makes it possible to introduce new binding types for new types of services.

B.3.1 Remotable Service

A Remotable Service is a service that is designed to be published remotely in a loosely-coupled SOA architecture. For example, SCA services of SCA implementations can define implementations of industry-standard web services. Remotable services use pass-by-value semantics for parameters and returned results.
A service is remotable if it is defined by a WSDL port type or if it defined by a Java interface marked with the @Remotable annotation.

B.3.2 Local Service

Local services are services that are designed to be only used “locally” by other implementations that are deployed concurrently in a tightly-coupled architecture within the same operating system process.

Local services may rely on by-reference calling conventions, or may assume a very fine-grained interaction style that is incompatible with remote distribution. They may also use technology-specific data-types.

Currently a service is local only if it defined by a Java interface not marked with the @Remotable annotation.

B.4 Reference

SCA references represent a dependency that an implementation has on a service that is supplied by some other implementation, where the service to be used is specified through configuration. In other words, a reference is a service that an implementation may call during the execution of its business function. References are typed by an interface.

For composites, composite references can be accessed by components within the composite like any service provided by a component within the composite. Composite references can be used as the targets of wires from component references when configuring Components.

A composite reference can be used to access a service such as: an SCA service provided by another SCA composite, a Web service, a stateless session EJB, a data base stored procedure or an EIS service***,*** and so on. References use ***bindings*** to describe the access method used to their services. SCA provides an ***extensibility mechanism*** that allows the introduction of new binding types to references.

B.5 Implementation

An implementation is concept that is used to describe a piece of software technology such as a Java class, BPEL process, XSLT transform, or C++ class that is used to implement one or more services in a service-oriented application. An SCA composite is also an implementation.

Implementations define points of variability including properties that can be set and settable references to other services. The points of variability are configured by a component that uses the implementation. The specification refers to the configurable aspects of an implementation as its ***componentType***.

B.6 Interface

Interfaces define one or more business functions. These business functions are provided by Services and are used by components through References. Services are defined by the Interface they implement. SCA currently supports two interface type systems:

- Java interfaces
- WSDL portTypes

SCA also provides an extensibility mechanism by which an SCA runtime can add support for additional interface type systems.

Interfaces may be ***bi-directional***. A bi-directional service has service operations which must be provided by each end of a service communication – this could be the case where a particular service requires a “callback” interface on the client, which is calls during the process of handing service requests from the client.

B.7 Composite

An SCA composite is the basic unit of composition within an SCA Domain. An ***SCA Composite*** is an assembly of Components, Services, References, and the Wires that interconnect them. Composites can be used to contribute elements to an ***SCA Domain***.

A ***composite*** has the following characteristics:

- It may be used as a component implementation. When used in this way, it defines a boundary for Component visibility. Components may not be directly referenced from outside of the composite in which they are declared.
- It can be used to define a unit of deployment. Composites are used to contribute business logic artifacts to an SCA domain.

B.8 Composite inclusion

One composite can be used to provide part of the definition of another composite, through the process of inclusion. This is intended to make team development of large composites easier. Included composites are merged together into the using composite at deployment time to form a single logical composite.

Composites are included into other composites through <include…/> elements in the using composite. The SCA Domain uses composites in a similar way, through the deployment of composite files to a specific location.

B.9 Property

Properties allow for the configuration of an implementation with externally set data values. The data value is provided through a Component, possibly sourced from the property of a containing composite.

Each Property is defined by the implementation. Properties may be defined directly through the implementation language or through annotations of implementations, where the implementation language permits, or through a componentType file. A Property can be either a simple data type or a complex data type. For complex data types, XML schema is the preferred technology for defining the data types.

B.10 Domain

An SCA Domain represents a set of Services providing an area of Business functionality that is controlled by a single organization. As an example, for the accounts department in a business, the SCA Domain might cover all finance-related functions, and it might contain a series of composites dealing with specific areas of accounting, with one for Customer accounts, another dealing with Accounts Payable.

A domain specifies the instantiation, configuration and connection of a set of components, provided via one or more composite files. The domain, like a composite, also has Services and References. Domains also contain Wires which connect together the Components, Services and References.

B.11 Wire

SCA wires connect ***service references*** to ***services***.

Within a composite, valid wire sources are component references and composite services. Valid wire targets are component services and composite references.

When using included composites, the sources and targets of the wires don't have to be declared in the same composite as the composite that contains the wire. The sources and targets can be defined by other included composites. Targets can also be external to the SCA domain.

C. Acknowledgements

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

Participants:

[Participant Name, Affiliation | Individual Member]

[Participant Name, Affiliation | Individual Member]

D. Non-Normative Text

E. Revision History

[optional; should not be included in OASIS Standards]

Revision	Date	Editor	Changes Made
1	2007-09-24	Anish Karmarkar	Applied the OASIS template + related changes to the Submission