Service Component Architecture Assembly Model Specification Version 1.1

Committee Draft 02

14th January 2009

Specification URIs:
This Version:

http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd02.html
http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd02.doc
http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd02.pdf (Authoritative)

Previous Version:

Latest Version:

http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec.html
http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec.doc
http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec.pdf (Authoritative)

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):

http://docs.oasis-open.org/ns/opencsa/sca/200712

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

OASIS requests that any OASIS Party or any other party that believes it has patent claims that would necessarily be infringed by implementations of this OASIS Committee Specification or OASIS Standard, to notify OASIS TC Administrator and provide an indication of its willingness to grant patent licenses to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification.

OASIS invites any party to contact the OASIS TC Administrator if it is aware of a claim of ownership of any patent claims that would necessarily be infringed by implementations of this specification by a patent holder that is not willing to provide a license to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification. OASIS may include such claims on its website, but disclaims any obligation to do so.

OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS' procedures with respect to rights in any document or deliverable produced by an OASIS Technical Committee can be found on the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this OASIS Committee Specification or OASIS Standard, can be obtained from the OASIS TC Administrator. OASIS makes no representation that any information or list of intellectual property rights will at any time be complete, or that any claims in such list are, in fact, Essential Claims.

The names "OASIS", [insert specific trademarked names and abbreviations here] are trademarks of OASIS, the owner and developer of this specification, and should be used only to refer to the organization and its official outputs. OASIS welcomes reference to, and implementation and use of, specifications, while reserving the right to enforce its marks against misleading uses. Please see http://www.oasis-open.org/who/trademark.php for above guidance.

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.

This specification is defined in terms of Infoset and not in terms of XML 1.0, even though the specification uses XML 1.0 terminology. A mapping from XML to infoset is trivial and should be used for any non-XML serializations.

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.

[SCA-Java] SCA Java Component Implementation Specification
http://www.osoa.org/download/attachments/35/SCA_JavaComponentImplementation_V100.pdf

[SCA-Common-Java] SCA Java Common Annotations and APIs Specification
http://www.osoa.org/download/attachments/35/SCA_JavaAnnotationsAndAPIs_V100.pdf

[SCA BPEL] SCA BPEL Client and Implementation Specification
http://docs.oasis-open.org/opencsa/sca-bpel/sca-bpel-1.1-spec-cd-01.pdf

[SDO] 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://docs.oasis-open.org/opencsa/sca-bindings/sca-wsbinding-1.1-spec-cd01.pdf

[10] SCA Policy Framework Specification
http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd-01.pdf

[11] SCA JMS Binding Specification
http://docs.oasis-open.org/opencsa/sca-bindings/sca-jmsbinding-1.1-spec-cd01.pdf

[SCA-CPP-Client] SCA C++ Client and Implementation Specification
http://docs.oasis-open.org/opencsa/sca-c-cpp/sca-cppcni-1.1-spec-cd-01.pdf

[SCA-C-Client] SCA C Client and Implementation Specification
http://docs.oasis-open.org/opencsa/sca-c-cpp/sca-ccni-1.1-spec-cd-01.pdf

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

[13] Infoset Specification
http://www.w3.org/TR/xml-infoset/

[WSDL11_Identifiers] WSDL 1.1 Element Identiifiers
http://www.w3.org/TR/wsdl11elementidentifiers/

1.3 Naming Conventions

This specification follows some naming conventions for artifacts defined by the specification, as follows:

- For the names of elements and the names of attributes within XSD files, the names follow the CamelCase convention, with all names starting with a lower case letter.
  eg <element name="componentType" type="sca:ComponentType"/>
- For the names of types within XSD files, the names follow the CamelCase convention with all names starting with an upper case letter.
  eg. <complexType name="ComponentService">
- For the names of intents, the names follow the CamelCase convention, with all names starting with a lower case letter, EXCEPT for cases where the intent represents an established acronym, in which case the entire name is in upper case.
  An example of an intent which is an acronym is the "SOAP" intent.

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 can 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 might 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 might 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:

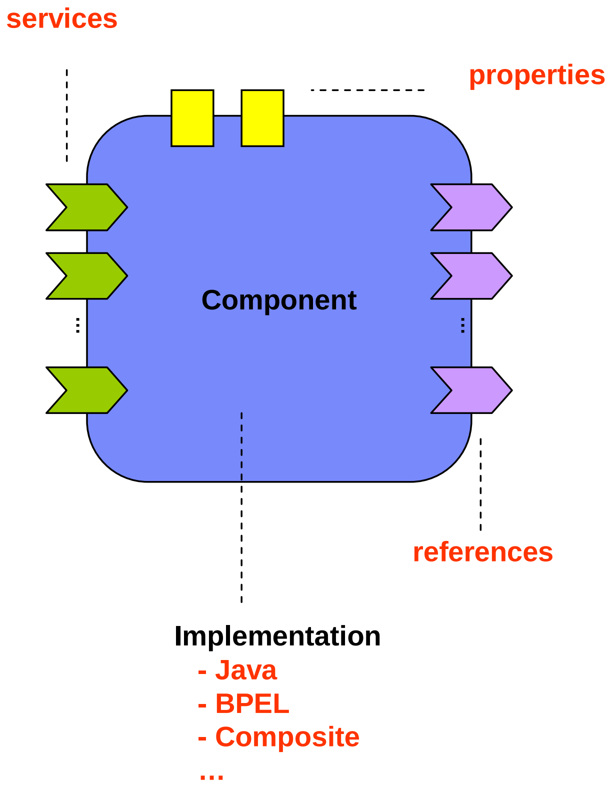
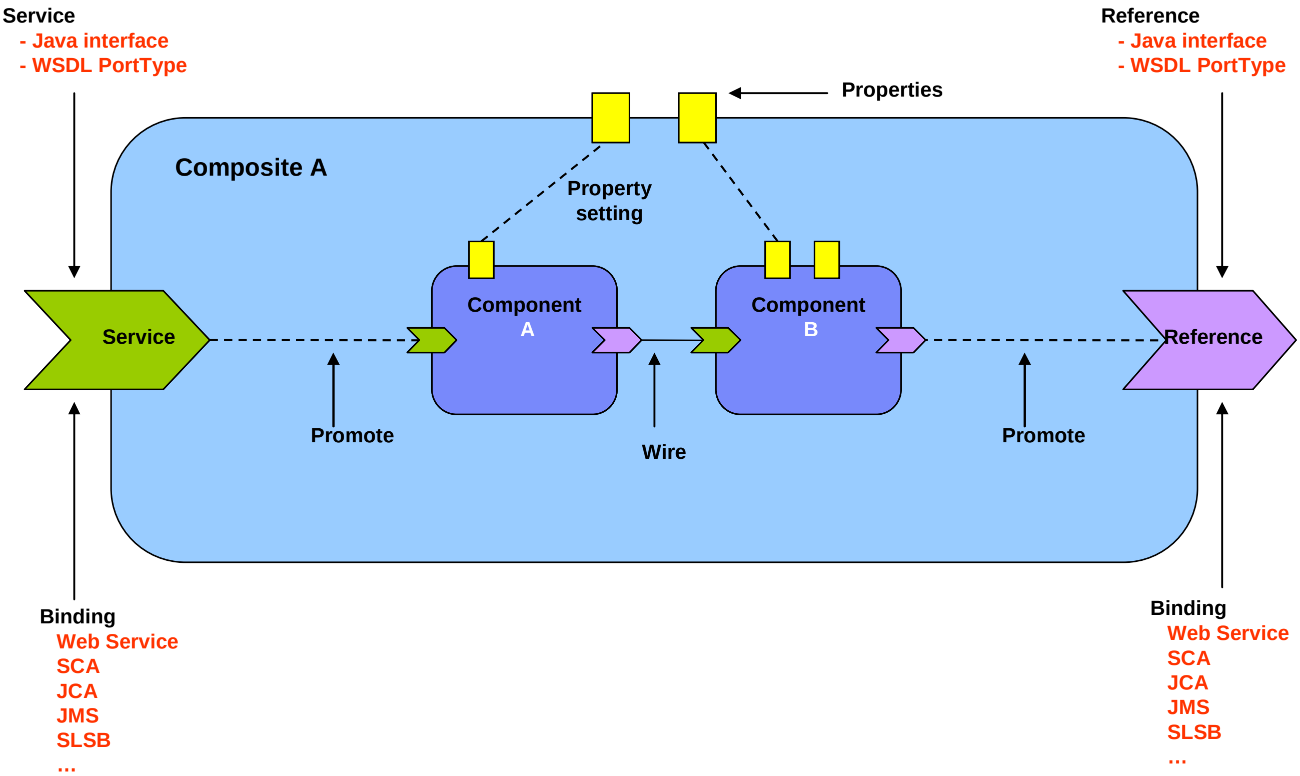


Figure 1: SCA Component Diagram

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

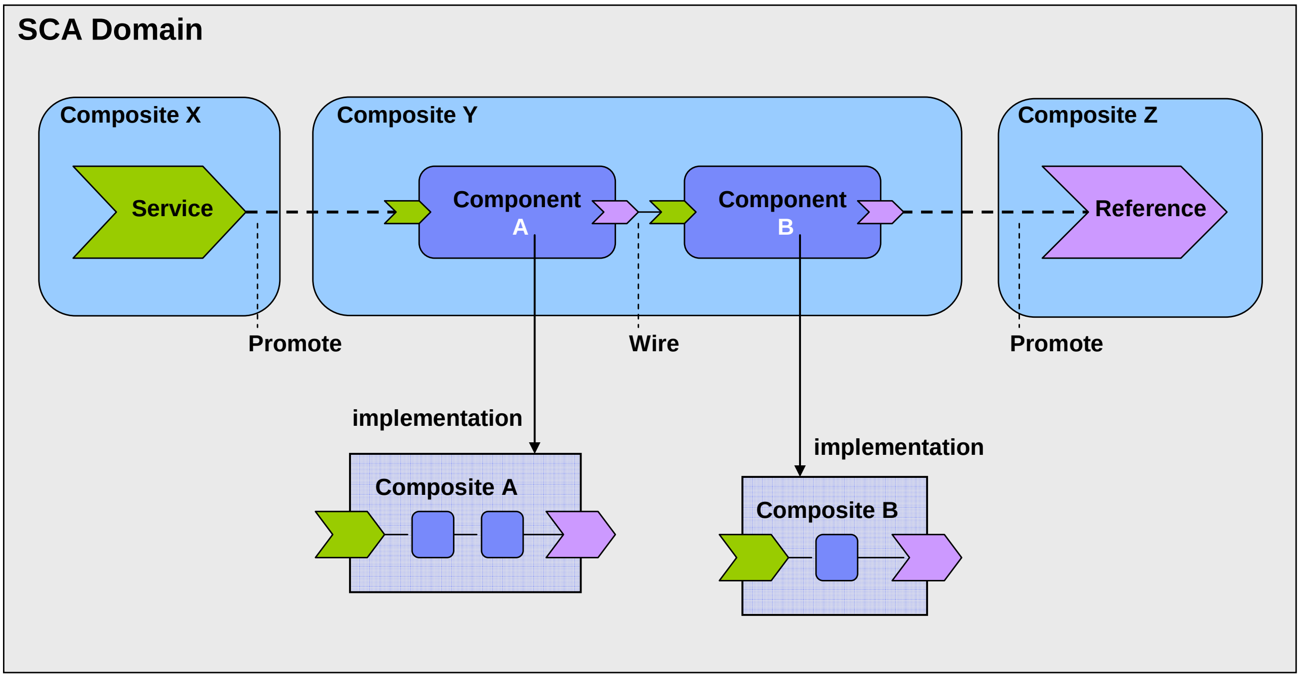


153

154 *Figure 2: SCA Composite Diagram*

155

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



158

159 *Figure 3: SCA Domain Diagram*

3 Quick Tour by Sample

To be completed.

This section is intended to contain a sample which describes the key concepts of SCA.

4 Implementation and ComponentType

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

SCA allows a choice of 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 might not simply define the implementation language, such as Java, but might also define the use of a specific framework or runtime environment. Examples include SCA Composite, Java implementations done using the Spring framework or the Java EE EJB technology.

Services, references and properties are the ***configurable aspects of an implementation***. SCA refers to them collectively as the ***component type***.

Depending on the implementation type, the implementation can declare the services, references and properties that it has and it also might be able to set values for all the characteristics of those services, references and properties.

So, for example:

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

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.

Most of the characteristics of the services, references and properties can 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).

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 that uses the implementation.

An implementation type specification (for example, the WS-BPEL Client and Implementation Specification Version 1.1 [SCA BPEL]) specifies the mechanism(s) by which the component type associated with an implementation of that type is derived.

Since SCA allows a broad range of implementation technologies, it is expected that some implementation technologies (for example, the Java Component Implementation Specification Version 1.1 [SCA-Java]) allow for introspecting the implementation artifact(s) (for example, a Java class) to derive the component type information. Other implementation technologies might not allow for introspection of the implementation artifact(s). In those cases where introspection is not allowed, SCA encourages the use of a SCA component type side file. A ***component type side file*** is an XML file whose document root element is sca:componentType.

The implementation type specification defines whether introspection is allowed, whether a side file is allowed, both are allowed or some other mechanism specifies the component type. The component type information derived through introspection is called the ***introspected component type***. In any case, the implementation type specification specifies how multiple sources of information are combined to produce the ***effective component type***. The effective component type is the component type metadata that is presented to the using Component for configuration.

The extension of a componentType side file name MUST be .componentType. [ASM40001] The name and location of a componentType side file, if allowed, is defined by the implementation type specification.

If a component type side file is not allowed for a particular implementation type, the effective component type and introspected component type are one and the same for that implementation type.

For the rest of this document, when the term 'component type' is used it refers to the 'effective component type'.

The following snippet shows the componentType pseudo-schema:

```
<?xml version="1.0" encoding="ASCII"?>
<!-- Component type schema snippet -->
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
               constrainingType="QName"? >

   <service … />*
   <reference … />*
   <property … />*
   <implementation … />?

</componentType>
```

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

- ***constrainingType : QName (0..1)*** – If present, the @constrainingType attribute of a <componentType/> element MUST reference a <constrainingType/> element in the Domain through its QName. [ASM40002] 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.

The ***componentType*** element has the following ***child elements***:

- ***service : Service (0..n)*** – see component type service section.
- ***reference : Reference (0..n)*** – see component type reference section.
- ***property : Property (0..n)*** – see component type property section.
- ***implementation : Implementation (0..1)*** – see component type implementation section.

4.1.1 Service

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. The following snippet shows the component type schema with the schema for a service child element:

```
<?xml version="1.0" encoding="ASCII"?>
<!-- Component type service schema snippet -->
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" …
>

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

   <reference … />*
   <property … />*
   <implementation … />?

</componentType>
```

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

- ***name : NCName (1..1)*** - the name of the service. The @name attribute of a <service/> child element of a <componentType/> MUST be unique amongst the service elements of that <componentType/>. [ASM40003]
- ***requires : QName (0..n)*** - a list of policy intents. See the Policy Framework specification [10] for a description of this attribute.
- ***policySets : QName (0..n)*** - a list of policy sets. See the Policy Framework specification [10] for a description of this attribute.

The ***service*** element has the following ***child elements***:

- ***interface : Interface (1..1)*** - A service has ***one interface***, which describes the operations provided by the service. For details on the interface element see the Interface section.
- ***operation: Operation (0..n)*** - Zero or more operation elements. These elements are used to describe characteristics of individual operations within the interface. For a detailed decription of the operation element, see the Policy Framework specification [SCA Policy].
- ***binding : Binding (0..n)*** - A service element has ***zero or more binding elements*** as children. If the binding element is not present it defaults to <binding.sca>. Details of the binding element are described in the Bindings section.
- ***callback (0..1) / binding : Binding (1..n)*** - 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. For details on callbacks, see the Bidirectional Interfaces section.

4.1.2 Reference

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. The following snippet shows the component type schema with the schema for a reference child element:

```
<?xml version="1.0" encoding="ASCII"?>
<!-- Component type reference schema snippet -->
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" …
>

   <service … />*

   <reference name="xs:NCName"
            autowire="xs:boolean"?
            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 … />
         <operation name="xs:NCName" requires="list of xs:QName"?
            policySets="list of xs:QName"?/>*
         <binding … />*
         <callback>?
               <binding … />+
         </callback>
   </reference>

   <property … />*
   <implementation … />?

</componentType>
```

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

- ***name : NCName (1..1)*** - the name of the reference. The @name attribute of a <reference/> child element of a <componentType/> MUST be unique amongst the reference elements of that <componentType/>. [ASM40004]
- ***multiplicity : 0..1|1..1|0..n|1..n (0..1)*** - defines the number of wires that can connect the reference to target services. The multiplicity can have the following values
  - 0..1 – zero or one wire can have the reference as a source
  - 1..1 – one wire can have the reference as a source
  - 0..n - zero or more wires can have the reference as a source
  - 1..n – one or more wires can have the reference as a source

  If @multiplicity is not specified, the default value is "1..1".
- ***autowire : boolean (0..1)*** - whether the reference should be autowired, as described in the Autowire section. Default is false.
- ***wiredByImpl : boolean (0..1)*** - a boolean value, "false" by default. If set to "false", the reference is wired to the target(s) configured on the reference. 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 @wiredByImpl is set to "true", then any reference targets configured for this reference MUST be ignored by the runtime. [ASM40006] It is recommended that any references with @wiredByImpl = "true" are left unwired.

- ***requires : QName (0..n)*** - a list of policy intents. See the Policy Framework specification [10] for a description of this attribute.
- ***policySets : QName (0..n)*** - a list of policy sets. See the Policy Framework specification [10] for a description of this attribute.

The ***reference*** element has the following ***child elements***:

- ***interface : Interface (1..1)*** - A reference has ***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. For details on the interface element see the Interface section.
- ***operation: Operation (0..n)*** - Zero or more operation elements. These elements are used to describe characteristics of individual operations within the interface. For a detailed decription of the operation element, see the Policy Framework specification [SCA Policy].
- ***binding : Binding (0..n)*** - A reference element has ***zero or more binding elements*** as children. Details of the binding element are described in the Bindings section.

  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.
- ***callback (0..1) / binding : Binding (1..n)*** - 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. For details on callbacks, see the Bidirectional Interfaces section.

4.1.3 Property

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. The following snippet shows the component type schema with the schema for a reference child element:

```
<?xml version="1.0" encoding="ASCII"?>
<!-- Component type property schema snippet -->
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" …
>

   <service … />*
   <reference … >*

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

```

```
        <implementation … />?

    </componentType>
```

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

- ***name : NCName (1..1)*** - the name of the property. The @name attribute of a <property/> child element of a <componentType/> MUST be unique amongst the property elements of that <componentType/>. [ASM40005]
- one of ***(1..1)***:
  - ***type : QName*** - the type of the property defined as the qualified name of an XML schema type. The value of the property @type attribute MUST be the QName of an XML schema type. [ASM40007]
  - ***element : QName*** - 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. The value of the property @element attribute MUST be the QName of an XSD global element. [ASM40008]
- ***many : boolean (0..1)*** - (optional) whether the property is single-valued (false) or multi-valued (true). In the case of a multi-valued property, it is presented to the implementation as a collection of property values. If many is not specified, it takes a default value of false.
- ***mustSupply : boolean (0..1)*** - 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. If mustSupply is not specified, it takes a default value of false.
- ***file : anyURI (0..1)*** - a dereferencable URI to a file containing a value for the property.
- ***requires : QName (0..n)*** - a list of policy intents. See the Policy Framework specification [10] for a description of this attribute.
- ***policySets : QName (0..n)*** - a list of policy sets. See the Policy Framework specification [10] for a description of this attribute.

The property element can contain a default property value as its content. The form of the default property value is as described in the section on Component Property.

The value for a property is supplied to the implementation of a component at the time that the implementation is started. The implementation can choose to use the supplied value in any way that it chooses. In particular, the implementation can alter the internal value of the property at any time. However, if the implementation queries the SCA system for the value of the property, the value as defined in the SCA composite is the value returned.

The componentType property element can contain an SCA default value for the property declared by the implementation. However, the implementation can have a property which has an implementation defined default value, where the default value is not represented in the componentType. An example of such a default value is where the default value is computed at runtime by some code contained in the implementation. If a using component needs to control the value of a property used by an implementation, the component sets the value explicitly. The SCA runtime MUST ensure that any implementation default property value is replaced by a value for that property explicitly set by a component using that implementation. [ASM40009]

4.1.4 Implementation

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. The following snippet shows the component type schema with the schema for a implementation child element:

```
<?xml version="1.0" encoding="ASCII"?>
<!-- Component type implementation schema snippet -->
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" …
>

   <service … />*
   <reference … >*
   <property … />*

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

</componentType>
```

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

- ***requires : QName (0..n)*** - a list of policy intents. See the Policy Framework specification [10] for a description of this attribute.
- ***policySets : QName (0..n)*** - a list of policy sets. See the Policy Framework specification [10] for a description of this attribute.

4.2 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://docs.oasis-open.org/ns/opencsa/sca/200712">

   <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>
```

4.3 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 {

    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://docs.oasis-open.org/ns/opencsa/sca/200712"
               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 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://docs.oasis-open.org/ns/opencsa/sca/200712" … >
   …
   <component name="xs:NCName" autowire="xs:boolean"?
               requires="list of xs:QName"? policySets="list of xs:QName"?
               constrainingType="xs:QName"?>*
         <implementation … />?
         <service … />*
         <reference … />*
         <property … />*
   </component>
   …
</composite>
```

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

- ***name : NCName (1..1)*** – the name of the component. The @name attribute of a <component/> child element of a <composite/> MUST be unique amongst the component elements of that <composite/> [ASM50001]
- ***autowire : boolean (0..1)*** – whether contained component references should be autowired, as described in the Autowire section. Default is false.
- ***requires : QName (0..n)*** – a list of policy intents. See the Policy Framework specification [10] for a description of this attribute.
- ***policySets : QName (0..n)*** – a list of policy sets. See the Policy Framework specification [10] for a description of this attribute.
- ***constrainingType : QName (0..1)*** – 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.

The ***component*** element has the following ***child elements***:

- ***implementation : ComponentImplementation (0..1)*** – see component implementation section.
- ***service : ComponentService (0..n)*** – see component service section.
- ***reference : ComponentReference (0..n)*** – see component reference section.
- ***property : ComponentProperty (0..n)*** – see component property section.

5.1 Implementation

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.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Component Implementation schema snippet -->
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" … >
   …
   <component … >*
         <implementation … />?
         <service … />*
         <reference … />*
         <property … />*
   </component>
   …
</composite>
```

The component provides the extensibility point in the assembly model for different implementation types. The references to implementations of different types are expressed by implementation type specific implementation elements.

For example the elements ***implementation.java***, ***implementation.bpel***, ***implementation.cpp***, and ***implementation.c*** point to Java, BPEL, C++, and C 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="ans:MoneyTransferProcess"/>
```

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

New implementation types can be added to the model as described in the Extension Model section.

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.

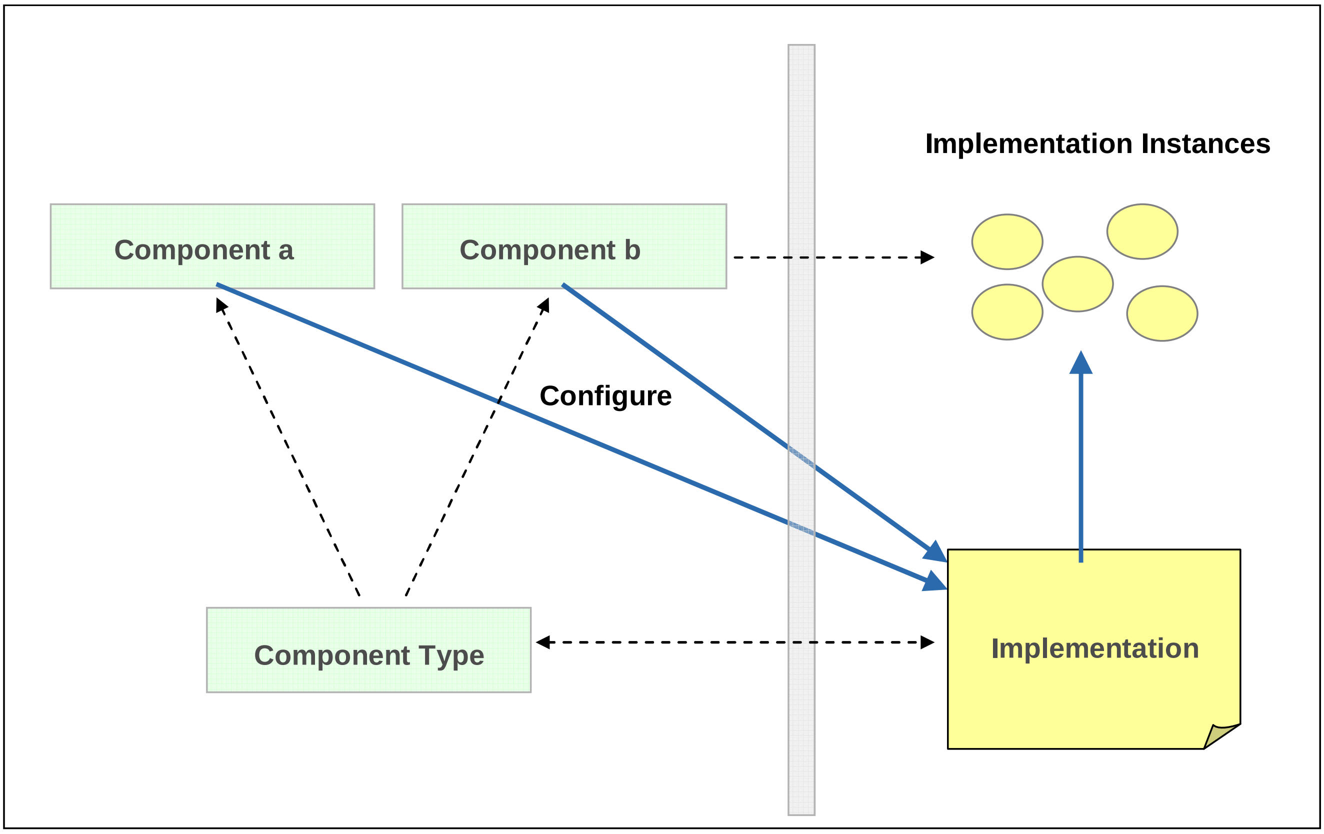


Figure 4: Relationship of Component and Implementation

5.2 Service

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 following snippet shows the component schema with the schema for a service child element:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Component Service schema snippet -->
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" … >
   …
   <component … >*
         <implementation … />?
         <service name="xs:NCName" requires="list of xs:QName"?
                  policySets="list of xs:QName"?>*
               <interface … />?
               <operation name="xs:NCName" requires="list of xs:QName"?
                  policySets="list of xs:QName"?/>*
               <binding … />*
               <callback>?
                     <binding … />+
         </callback>
         </service>
         <reference … />*
         <property … />*
   </component>
```

```
    …
</composite>
```

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

- ***name : NCName (1..1)*** - the name of the service. The @name attribute of a service element of a <component/> MUST be unique amongst the service elements of that <component/> [ASM50002] The @name attribute of a service element of a <component/> MUST match the @name attribute of a service element of the componentType of the <implementation/> child element of the component. [ASM50003]
- ***requires : QName (0..n)*** – 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 : QName (0..n)*** – a list of policy sets. See the Policy Framework specification [10] for a description of this attribute.

The ***component service*** element has the following ***child elements***:

- ***interface : Interface (0..1)*** - 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 in the componentType of the implementation is in effect. If a <service/> element has an interface subelement specified, the interface MUST provide a compatible subset of the interface declared on the componentType of the implementation [ASM50004] For details on the interface element see the Interface section.
- ***operation: Operation (0..n)*** - Zero or more operation elements. These elements are used to describe characteristics of individual operations within the interface. For a detailed decription of the operation element, see the Policy Framework specification [SCA Policy].
- ***binding : Binding (0..n)*** - A service element has ***zero or more binding elements*** as children. If no binding elements are specified for the service, then the bindings specified for the equivalent service in the componentType of the implementation MUST be used, but if the componentType also has no bindings specified, then <binding.sca/> MUST be used as the binding. If binding elements are specified for the service, then those bindings MUST be used and they override any bindings specified for the equivalent service in the componentType of the implementation. [ASM50005] Details of the binding element are described in the Bindings section. The binding, combined with any PolicySets in effect for the binding, needs to satisfy the set of policy intents for the service, as described in the Policy Framework specification [10].
- ***callback (0..1) / binding : Binding (1..n)*** - 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 present and contains one or more binding child elements, then those bindings MUST be used for the callback. [ASM50006] If the callback element is not present, the behaviour is runtime implementation dependent.

5.3 Reference

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 following snippet shows the component schema with the schema for a reference child element:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Component Reference schema snippet -->
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" … >
   …
   <component … >*
         <implementation … />?
         <service … />*
         <reference name="xs:NCName"
                  target="list of xs:anyURI"? autowire="xs:boolean"?
                  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 … />?
               <operation name="xs:NCName" requires="list of xs:QName"?
                  policySets="list of xs:QName"?/>*
               <binding uri="xs:anyURI"? requires="list of xs:QName"?
                  policySets="list of xs:QName"?/>*
               <callback>?
                     <binding … />+
               </callback>
         </reference>
         <property … />*
   </component>
   …
</composite>
```

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

- ***name : NCName (1..1)*** – the name of the reference. The @name attribute of a service element of a <component/> MUST be unique amongst the service elements of that <component/> [ASM50007] The @name attribute of a reference element of a <component/> MUST match the @name attribute of a reference element of the componentType of the <implementation/> child element of the component. [ASM50008]
- ***autowire : boolean (0..1)*** – whether the reference should be autowired, as described in the Autowire section. Default is false.
- ***requires : QName (0..n)*** – 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 : QName (0..n)*** – a list of policy sets. See the Policy Framework specification [10] for a description of this attribute.
- ***multiplicity : 0..1|1..1|0..n|1..n (0..1)*** - defines the number of wires that can connect the reference to target services. Overrides the multiplicity specified for this reference in the componentType of the implementation. The multiplicity can have the following values
  - 0..1 – zero or one wire can have the reference as a source
  - 1..1 – one wire can have the reference as a source
  - 0..n - zero or more wires can have the reference as a source
  - 1..n – one or more wires can have the reference as a source

  The value of multiplicity for a component reference MUST only be equal or further restrict any value for the multiplicity of the reference with the same name in the componentType of the implementation, where further restriction means 0..n to 0..1 or 1..n to 1..1. [ASM50009]

If not present, the value of multiplicity is equal to the multiplicity specificed for this reference in the componentType of the implementation - if not present in the componentType, the value defaults to 1..1.

- ***target : anyURI (0..n)*** – 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 : boolean (0..1)*** – 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 @wiredByImpl="true" is set for a reference, then the reference MUST NOT be wired statically within a composite, but left unwired. [ASM50010]

The ***component reference*** element has the following ***child elements***:

- ***interface : Interface (0..1)*** - 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 in the componentType of the implementation is in effect. If an interface is declared for a component reference it MUST provide a compatible superset of the interface declared for the equivalent reference in the componentType of the implementation, i.e. provide the same operations or a superset of the operations defined by the implementation for the reference. [ASM50011] For details on the interface element see the Interface section.
- ***operation: Operation (0..n)*** - Zero or more operation elements. These elements are used to describe characteristics of individual operations within the interface. For a detailed decription of the operation element, see the Policy Framework specification [SCA Policy].
- ***binding : Binding (0..n)*** - A reference element has ***zero or more binding elements*** as children. If no binding elements are specified for the reference, then the bindings specified for the equivalent reference in the componentType of the implementation MUST be used, but if the componentType also has no bindings specified, then <binding.sca/> MUST be used as the binding. If binding elements are specified for the reference, then those bindings MUST be used and they override any bindings specified for the equivalent reference in the componentType of the implementation. [ASM50012] Details of the binding element are described in the Bindings section. The binding, combined with any PolicySets in effect for the binding, needs to satisfy the set of policy intents for the reference, as described in the Policy Framework specification [10].

  A reference identifies zero or more target services that satisfy the reference. This can be done in a number of ways, which are fully described in section "5.3.1 Specifying the Target Service(s) for a Reference"
- ***callback (0..1) / binding : Binding (1..n)*** - 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 present and contains one or more binding child elements, then those bindings MUST be used for the callback. [ASM50006] If the callback element is not present, the behaviour is runtime implementation dependent.

5.3.1 Specifying the Target Service(s) for a Reference

A reference defines zero or more target services that satisfy the reference. The target service(s) can be defined in the following ways:

1. Through a value specified in the @target attribute of the reference element

2. Through a target URI specified in the @uri attribute of a binding element which is a child of the reference element
3. Through the setting of one or more values for binding-specific attributes and/or child elements of a binding element that is a child of the reference element
4. Through the specification of @autowire="true" for the reference (or through inheritance of that value from the component or composite containing the reference)
5. Through the specification of @wiredByImpl="true" for the reference
6. Through the promotion of a component reference by a composite reference of the composite containing the component (the target service is then identified by the configuration of the composite reference)
7. Through the presence of a <wire/> element which has the reference specified in its @source attribute.

Combinations of these different methods are allowed, and the following rules MUST be observed:

- If @wiredByImpl="true", other methods of specifying the target service MUST NOT be used. [ASM50013]
- If @autowire="true", the autowire procedure MUST only be used if no target is identified by any of the other ways listed above. It is not an error if @autowire="true" and a target is also defined through some other means, however in this case the autowire procedure MUST NOT be used. [ASM50014]
- If a reference has a value specified for one or more target services in its @target attribute, there MUST NOT be any child <binding/> elements declared for that reference. [ASM50026]
- If a binding element has a value specified for a target service using its @uri attribute, the binding element MUST NOT identify target services using binding specific attributes or elements. [ASM50015]
- It is possible that a particular binding type MAY require that the address of a target service uses more than a simple URI. In such cases, the @uri attribute MUST NOT be used to identify the target service - instead, binding specific attributes and/or child elements must be used. [ASM50016]
- If any <wire/> element with its @replace attribute set to "true" has a particular reference specified in its @source attribute, the value of the @target attribute for that reference MUST be ignored and MUST NOT be used to define target services for that reference. [ASM50034]

5.3.1.1 Multiplicity and the Valid Number of Target Services for a Reference

The number of target services configured for a reference are constrained by the following rules.

- A reference with multiplicity 0..1 or 0..n MAY have no target service defined. [ASM50018]
- A reference with multiplicity 0..1 or 1..1 MUST NOT have more that one target service defined. [ASM50019]
- A reference with multiplicity 1..1 or 1..n MUST have at least one target service defined. [ASM50020]
- A reference with multiplicity 0..n or 1..n MAY have one or more target services defined. [ASM50021]

Where it is detected that the rules for the number of target services for a reference have been violated, either at deployment or at execution time, an SCA Runtime MUST generate an error no later than when the reference is invoked by the component implementation. [ASM50022]

Some reference multiplicity errors can be detected at deployment time. In these cases, an error SHOULD be generated by the SCA runtime at deployment time. [ASM50023] For example, where a composite is used as a component implementation, wires and target services cannot be added to the composite after deployment. As a result, for components which are part of the composite, both missing wires and wires with a non-existent target can be detected at deployment time through a scan of the contents of the composite.

Other reference multiplicity errors can only be checked at runtime. In these cases, the SCA runtime MUST generate an error no later than when the reference is invoked by the component implementation. [ASM50024] Examples include cases of components deployed to the SCA Domain. At the Domain level, the target of a wire, or even the wire itself, may form part of a separate deployed contribution and as a result these may be deployed after the original component is deployed. For the cases where it is valid for the reference to have no target service specified, the component implementation language specification needs to define the programming model for interacting with an untargetted reference.

Where a component reference is promoted by a composite reference, the promotion MUST be treated from a multiplicity perspective as providing 0 or more target services for the component reference, depending upon the further configuration of the composite reference. These target services are in addition to any target services identified on the component reference itself, subject to the rules relating to multiplicity. [ASM50025]

5.4 Property

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 component type of 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 five ways:

- As a value, supplied in the ***value*** attribute of the property element.
  If the @value attribute of a component property element is declared, the type of the property MUST be an XML Schema simple type and the @value attribute MUST contain a single value of that type. [ASM50027]

  For example,

  ```
  <property name="pi" value="3.14159265" />
  ```

- As a value, supplied as the content of the ***value*** element(s) children of the property element.
  If the value subelement of a component property is specified, the type of the property MUST be an XML Schema simple type or an XML schema complex type. [ASM50028]

  For example,

  - property defined using a XML Schema simple type and which contains a single value

    ```
    <property name="pi">
        <value>3.14159265</value>
    </property>
    ```

  - property defined using a XML Schema simple type and which contains multiple values

    ```
    <property name="currency">
        <value>EURO</value>
        <value>USDollar</value>
    </property>
    ```

  - property defined using a XML Schema complex type and which contains a single value

    ```
    <property name="complexFoo">
        <value attr="bar">
                <foo:a>TheValue</foo:a>
                <foo:b>InterestingURI</foo:b>
    ```

```
    </value>
</property>
```

    - property defined using a XML Schema complex type and which contains multiple values

```
<property name="complexBar">
    <value anotherAttr="foo">
            <bar:a>AValue</bar:a>
            <bar:b>InterestingURI</bar:b>
    </value>
    <value attr="zing">
            <bar:a>BValue</bar:a>
            <bar:b>BoringURI</bar:b>
    </value>
</property>
```

- As a value, supplied as the content of the property element.
  If a component property value is declared using a child element of the <property/> element, the type of the property MUST be an XML Schema global element and the declared child element MUST be an instance of that global element. [ASM50029]

  For example,

    - property defined using a XML Schema global element declartion and which contains a single value

```
<property name="foo">
    <foo:SomeGED ...>...</foo:SomeGED>
</property>
```

    - property defined using a XML Schema global element declaration and which contains multiple values

```
<property name="bar">
    <bar:SomeOtherGED ...>...</bar:SomeOtherGED>
    <bar:SomeOtherGED ...>...</bar:SomeOtherGED>
</property>
```

- 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 composite property is of a complex type, the XPath expression can be extended to refer to a sub-part of the complex property 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.

For a property defined using a XML Schema simple type and for which a single value is desired, can be set either using the @value attribute or the <value> child element. The two forms in such a case are equivalent.

When a property has multiple values set, they MUST all be contained within the same property element. A <component/> element MUST NOT contain two <property/> subelements with the same value of the @name attribute. [ASM50030]

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 in the component type of the implementation. If no type is declared in the component property, the type of the property declared by the implementation is used.

The following snippet shows the component schema with the schema for a property child element:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Component Property schema snippet -->
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" … >
   …
   <component … >*
         <implementation … />?
         <service … />*
         <reference … />*
         <property name="xs:NCName"
                     (type="xs:QName" | element="xs:QName")?
                     mustSupply="xs:boolean"? many="xs:boolean"?
                     source="xs:string"? file="xs:anyURI"?
                     requires="list of xs:QName"?
                     policySets="list of xs:QName"?
                     value="xs:string"?>*
               [<value>+ | xs:any+ ]?
         </property>
   </component>
   …
</composite>
```

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

- ***name : NCName (1..1)*** – the name of the property. The name attribute of a component property MUST match the name of a property element in the component type of the component implementation. [ASM50031]
- zero or one of ***(0..1)***:
    - ***type : QName*** – the type of the property defined as the qualified name of an XML schema type
    - ***element : QName*** – 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 : string (0..1)*** – an XPath expression pointing to a property of the containing composite from which the value of this component property is obtained.
- ***file : anyURI (0..1)*** – a dereferencable URI to a file containing a value for the property
- ***many : boolean (0..1)*** – (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. If many is not specified, it takes the value defined by the component type of the implementation used by the component.
- ***value : string (0..1)*** - the value of the property if the property is defined using a simple type.
- ***requires : QName (0..n)*** - a list of policy intents. See the Policy Framework specification [10] for a description of this attribute.

- ***policySets : QName (0..n)*** - a list of policy sets. See the Policy Framework specification [10] for a description of this attribute.

The ***component property*** element has the following ***child element***:

value :any (0..n) - A property has ***zero or more***, value elements that specify the value(s) of a property that is defined using a XML Schema type. If a property is single-valued, the <value/> subelement MUST NOT occur more than once. [ASM50032] A property <value/> subelement MUST NOT be used when the @value attribute is used to specify the value for that property. [ASM50033]

5.5 Example Component

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

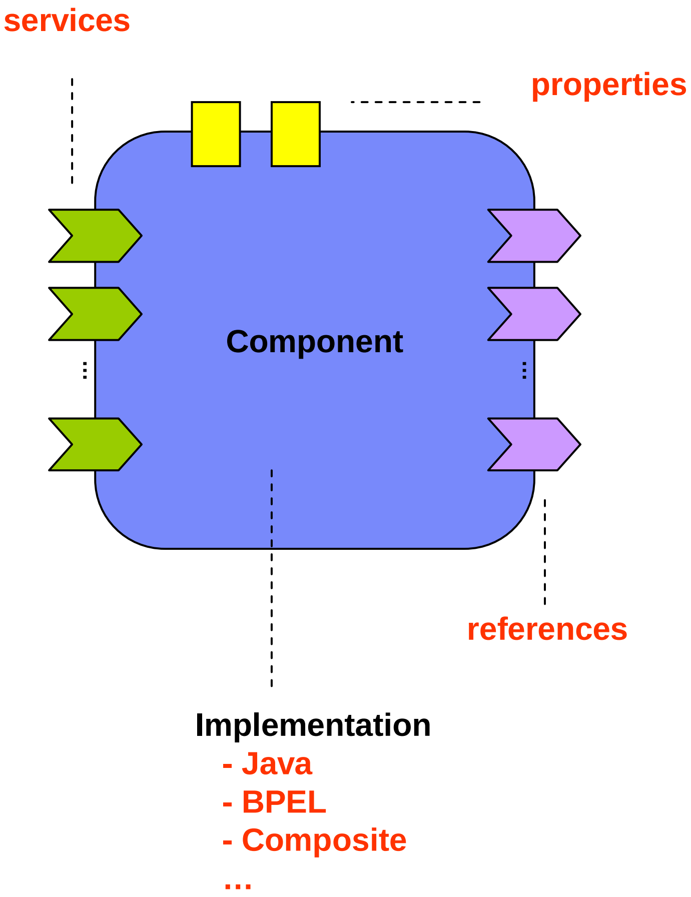


Figure 5: Component symbol

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

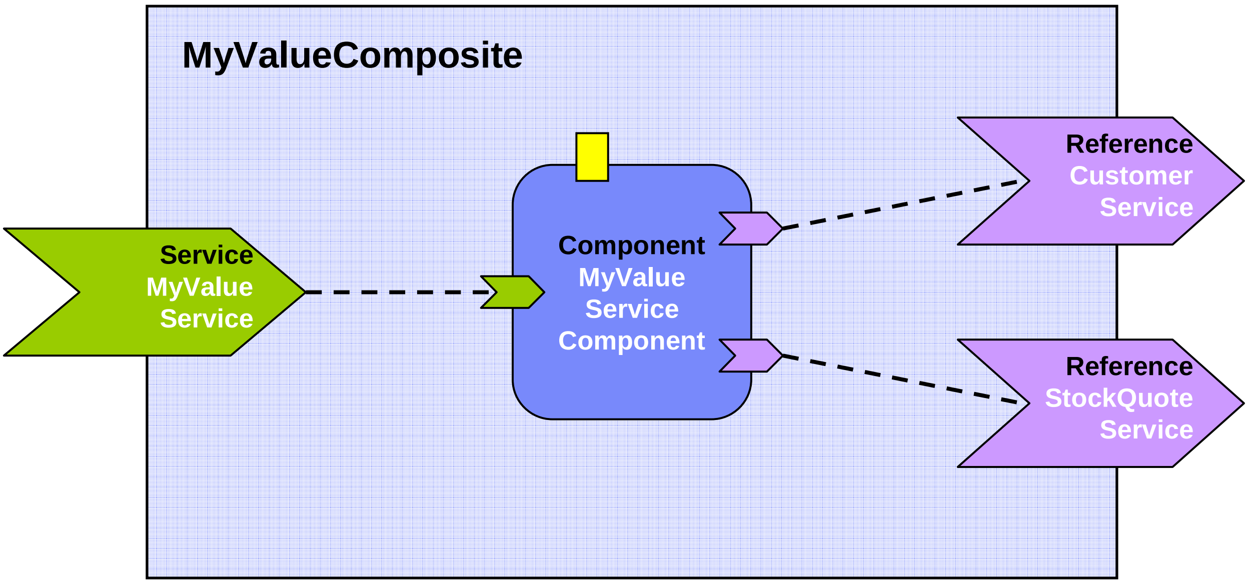


Figure 6: Assembly diagram for MyValueComposite

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://docs.oasis-open.org/ns/opencsa/sca/200712"
               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://docs.oasis-open.org/ns/opencsa/sca/200712"
               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.

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 can be used as ***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 can 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://docs.oasis-open.org/ns/opencsa/sca/200712"
         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 … />*
   <service … />*
   <reference … />*
   <property … />*
   <component … />*
   <wire … />*

</composite>
```

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

- ***name : NCName (1..1)*** – the name of the composite. The form of a composite name is an XML QName, in the namespace identified by the targetNamespace attribute. A composite name must be unique within the namespace of the composite. [ASM60001]
- ***targetNamespace : anyURI (0..1) –*** an identifier for a target namespace into which the composite is declared
- ***local : boolean (0..1)*** – whether all the components within the composite all run in the same operating system process. @local="true" for a composite means that all the components within the composite MUST run in the same operating system process. [ASM60002] local="false", which is the default, means that different components within the composite can run in different operating system processes and they can even run on different nodes on a network.

- ***autowire : boolean (0..1)*** – whether contained component references should be autowired, as described in the Autowire section. Default is false.
- ***constrainingType : QName (0..1)*** – 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 : QName (0..n)*** – a list of policy intents. See the Policy Framework specification [10] for a description of this attribute.
- ***policySets : QName (0..n)*** – a list of policy sets. See the Policy Framework specification [10] for a description of this attribute.

The ***composite*** element has the following ***child elements***:

- ***service : CompositeService (0..n)*** – see composite service section.
- ***reference : CompositeReference (0..n)*** – see composite reference section.
- ***property : CompositeProperty (0..n)*** – see composite property section.
- ***component : Component (0..n)*** – see component section.
- ***wire : Wire (0..n)*** – see composite wire section.
- ***include : Include (0..n)*** – see composite include section

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 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 pseudo-schema for a service child element:

```
<?xml version="1.0" encoding="ASCII"?>
<!-- Composite Service schema snippet -->
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" … >
   …
   <service name="xs:NCName" promote="xs:anyURI"
             requires="list of xs:QName"? policySets="list of xs:QName"?>*
         <interface … />?
         <operation name="xs:NCName" requires="list of xs:QName"?
            policySets="list of xs:QName"?/>*
         <binding … />*
         <callback>?
               <binding … />+
         </callback>
   </service>
   …
</composite>
```

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

- ***name : NCName (1..1)*** – the name of the service.The name of a composite <service/> element MUST be unique across all the composite services in the composite. [ASM60003] The name of the composite service can be different from the name of the promoted component service.
- ***promote : anyURI (1..1)*** – 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. The same component service can be promoted by more then one composite service. A composite <service/> element's promote attribute MUST identify one of the component services within that composite. [ASM60004]
- ***requires : QName (0..n)*** – a list of required policy intents. See the Policy Framework specification [10] for a description of this attribute. Specified ***required intents*** add to or further qualify the required intents defined by the promoted component service.
- ***policySets : QName (0..n)*** – a list of policy sets. See the Policy Framework specification [10] for a description of this attribute.

The ***composite service*** element has the following ***child elements***, whatever is not specified is defaulted from the promoted component service.

- ***interface : Interface (0..1)*** - If a composite service ***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. [ASM60005] 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.
- ***operation: Operation (0..n)*** - Zero or more operation elements. These elements are used to describe characteristics of individual operations within the interface. For a detailed decription of the operation element, see the Policy Framework specification [SCA Policy].
- ***binding : Binding (0..n)*** - 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.

- ***callback (0..1) / binding : Binding (1..n)*** - 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.

6.1.1 Service Examples

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

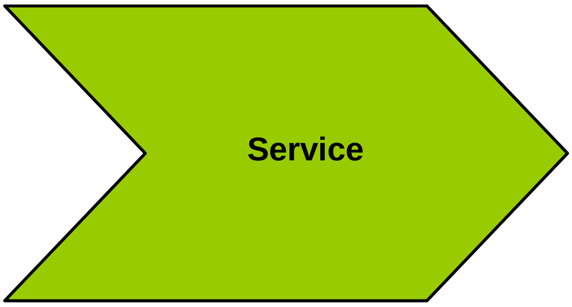


Figure 7: Service symbol

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

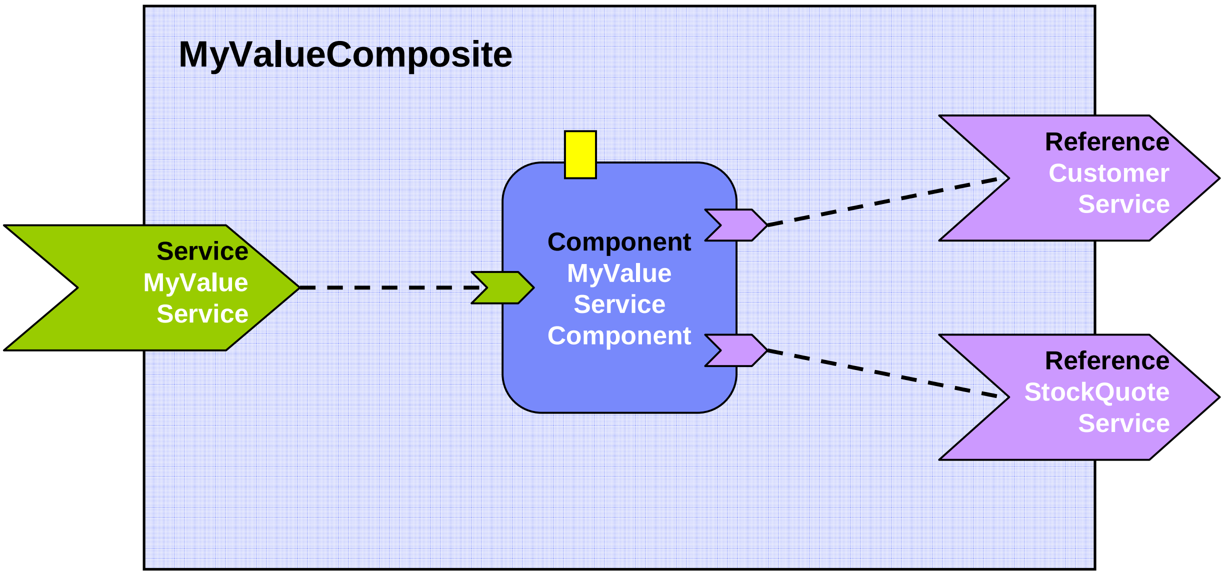

MyValueComposite
Service MyValue Service
Component MyValue Service Component
Reference Customer Service
Reference StockQuote Service

Figure 8: 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://docs.oasis-open.org/ns/opencsa/sca/200712"
               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>
```

6.2 Reference

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 needs to 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"?>
<!-- Composite Reference schema snippet -->
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" … >
   …
   <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 … />?
         <operation name="xs:NCName" requires="list of xs:QName"?
            policySets="list of xs:QName"?/>*
         <binding … />*
         <callback>?
               <binding … />+
        </callback>
   </reference>
   …
```

```
</composite>
```

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

- ***name : NCName (1..1)*** – the name of the reference. The name of a composite <reference/> element MUST be unique across all the composite references in the composite. [ASM60006] The name of the composite reference can be different then the name of the promoted component reference.
- ***promote : anyURI (1..n)*** – 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. Each of the URIs declared by a composite reference's @promote attribute MUST identify a component reference within the composite. [ASM60007]

  The same component reference can be 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.

  Where a composite reference promotes two or more component references:

  - the interfaces of the component references promoted by a composite reference MUST be the same, or if the composite reference itself declares an interface then all the component reference interfaces must be compatible with the composite reference interface. Compatible means that the component reference interface is the same or is a strict subset of the composite reference interface. [ASM60008]
  - the intents declared on a composite reference and on the component references which it promoites MUST NOT be mutually exclusive. [ASM60009] 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 plus any intents declared on the composite reference itself. If any intents in the set which apply to a composite reference are mutually exclusive then the SCA runtime MUST raise an error. [ASM60010]
- ***requires : QName (0..n)*** – a list of required policy intents. See the Policy Framework specification [10] for a description of this attribute. Specified ***required intents*** add to or further qualify the required intents defined for the promoted component reference.
- ***policySets : QName (0..n)*** – a list of policy sets. See the Policy Framework specification [10] for a description of this attribute.
- ***multiplicity : 0..1|1..1|0..n|1..n (1..1)*** - Defines the number of wires that can connect the reference to target services. The multiplicity can have the following values
  - 0..1 – zero or one wire can have the reference as a source
  - 1..1 – one wire can have the reference as a source
  - 0..n - zero or more wires can have the reference as a source
  - 1..n – one or more wires can have the reference as a source

  The value specified for the ***multiplicity*** attribute of a composite reference MUST be compatible with the multiplicity specified on each of the promoted component references, i.e. the multiplicity has to be equal or further restrict. So multiplicity 0..1 can be used where the promoted component reference has multiplicity 0..n, multiplicity 1..1 can be used where the promoted component reference has multiplicity 0..n or 1..n and multiplicity 1..n can be used where the promoted component reference has multiplicity 0..n., 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. [ASM60011]
- ***target : anyURI (0..n)*** – 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 : boolean (0..1)*** – 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*** element has the following ***child elements***, whatever is not specified is defaulted from the promoted component reference(s).

- ***interface : Interface (0..1)*** - ***zero or one interface element*** which declares an interface for the composite reference. If a composite reference has an ***interface*** specified, it MUST provide an interface which is the same or which is a compatible superset of the interface(s) declared by the promoted component reference(s), i.e. provide a superset of the operations in the interface defined by the component for the reference. [ASM60012] If no interface is declared on a composite reference, the interface from one of its promoted component references is used, which MUST be the same as or a compatible superset of the interface(s) declared by the promoted component reference(s). [ASM60013] For details on the interface element see the Interface section.
- ***operation: Operation (0..n)*** - Zero or more operation elements. These elements are used to describe characteristics of individual operations within the interface. For a detailed decription of the operation element, see the Policy Framework specification [SCA Policy].
- ***binding : Binding (0..n)*** - A reference element has zero or more ***binding elements*** as children. 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. Details of the binding element are described in the Bindings section. For more details on wiring see the section on Wires.

  A reference identifies zero or more target services which satisfy the reference. This can be done in a number of ways, which are fully described in section "5.3.1 Specifying the Target Service(s) for a Reference".
- ***callback (0..1) / binding : Binding (1..n)*** - 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.

6.2.1 Example Reference

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

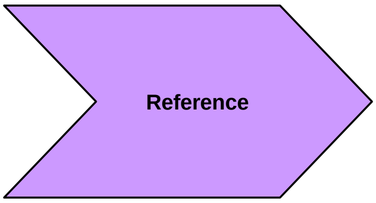


Figure 9: Reference symbol

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

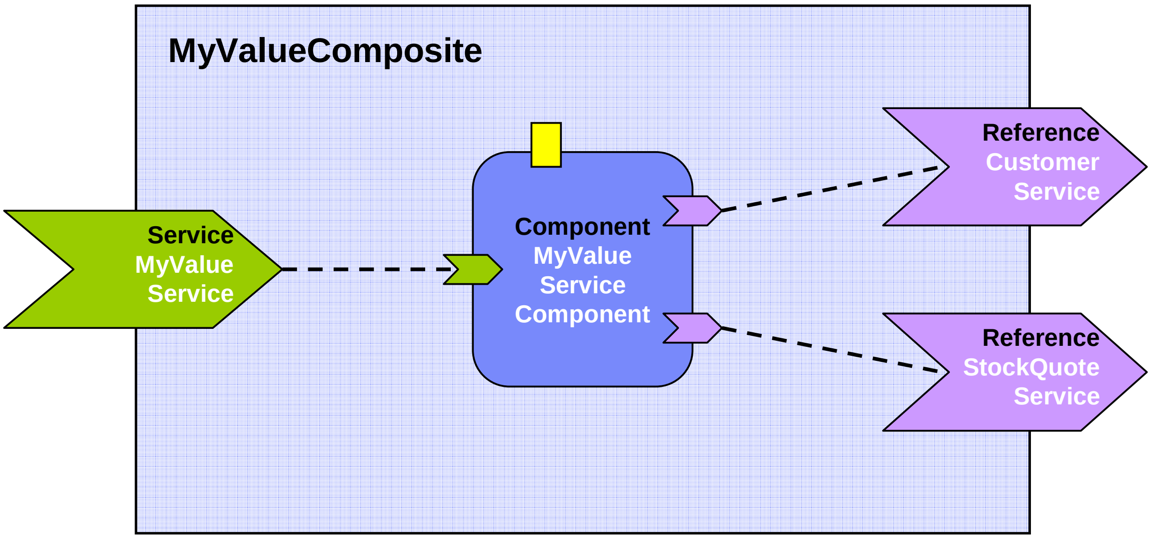

MyValueComposite
Service MyValue Service
Component MyValue Service Component
Reference Customer Service
Reference StockQuote Service

Figure 10: 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://docs.oasis-open.org/ns/opencsa/sca/200712"
               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      -->
         <!-- whaever 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 Property

Properties allow for the configuration of an implementation with externally set data values. A composite can declare zero or more properties. Each property has a type, which may be either simple or complex. An implementation can also define a default value for a property. Properties can be 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 Property schema snippet -->
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" … >
   …
   <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
                     requires="list of xs:QName"?
                     policySets="list of xs:QName"?
                     many="xs:boolean"? mustSupply="xs:boolean"?>*
         default-property-value?
   </property>
   …
</composite>
```

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

- ***name : NCName (1..1)*** - the name of the property. The name attribute of a composite property MUST be unique amongst the properties of the same composite. [ASM60014]
- one of ***(1..1)***:
  - ***type : QName*** – the type of the property - the qualified name of an XML schema type
  - ***element : QName*** – 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 : boolean (0..1)*** - 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 : boolean (0..1)*** – whether the property value has to be supplied by the component that uses the composite – when mustSupply="true" the component has to supply a value since the composite has no default value for the property. A default-property-value is only worth declaring 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.

- ***requires : QName (0..n)*** - a list of policy intents. See the Policy Framework specification [10] for a description of this attribute.
- ***policySets : QName (0..n)*** - a list of policy sets. See the Policy Framework specification [10] for a description of this attribute.

The property element may contain an optional ***default-property-value***, which provides default value for the property. The form of the default property value is as described in the section on Component Property.

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.

6.3.1 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://docs.oasis-open.org/ns/opencsa/sca/200712"
               xmlns:foo="http://foo.com"
               targetNamespace="http://foo.com"
               name="AccountServices">
<!-- AccountServices Example1 -->

   ...

   <property name="complexFoo" type="foo:MyComplexType">
         <value>
               <foo:a>AValue</foo:a>
               <foo:b>InterestingURI</foo:b>
         </value>
   </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 ***value*** which is required to be 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://docs.oasis-open.org/ns/opencsa/sca/200712"
               xmlns:foo="http://foo.com"
               targetNamespace="http://foo.com"
               name="AccountServices">
<!-- AccountServices Example2 -->

   ...

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

```

Declaration of a property with a global element type:

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

```

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 ...>
   ...
   <wire source="xs:anyURI" target="xs:anyURI" replace="xs:boolean"?/>*
   ...
</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 (1..1)*** – 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 (1..1)*** – 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
- ***replace (0..1)*** - a boolean value, with the default of "false". When a wire element has @replace="false", the wire is added to the set of wires which apply to the reference identified by the @source attribute. When a wire element has @replace="true", the wire is added to the set of wires which apply to the reference identified by the @source attribute - but any wires for that reference specified by means of the @target attribute of the reference are removed from the set of wires which apply to the reference.

  In other words, if any <wire/> element with @replace="true" is used for a particular reference, the value of the @target attribute on the reference is ignored - and this permits existing wires on the reference to be overridden by separate configuration, if required, where the reference is on a component at the Domain level.

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 of a wire MUST either both be remotable or else both be local [ASM60015]
2. the operations on the target interface of a wire MUST be the same as or be a superset of the operations in the interface specified on the source [ASM60016]
3. compatibility between the source interface and the target interface for a wire for the individual operations is defined as compatibility of the signature, that is operation name, input types, and output types MUST be the same. [ASM60017]

4. the order of the input and output types for operations in the source interface and the target interface of a wire also MUST be the same. [ASM60018]

5. the set of Faults and Exceptions expected by each operation in the source interface MUST be the same or be a superset of those specified by the target interface. [ASM60019]

6. other specified attributes of the source interface and the target interface of a wire MUST match, including Scope and Callback interface [ASM60020]

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. [ASM60021]

6.4.1 Wire Examples

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

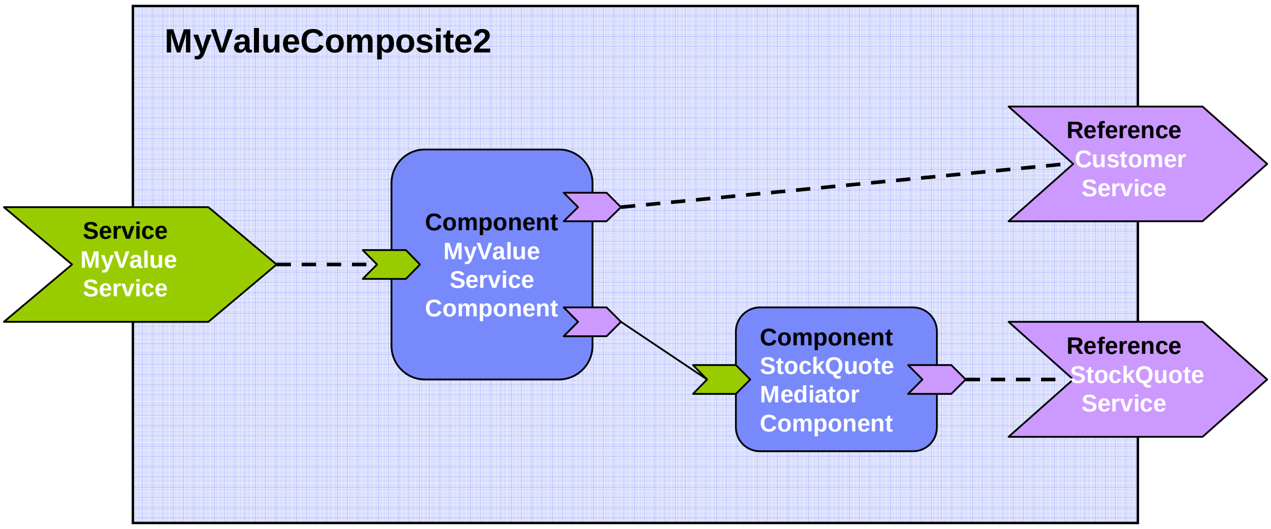

MyValueComposite2
Reference Customer Service
Service MyValue Service
Component MyValue Service Component
Component StockQuote Mediator Component
Reference StockQuote Service

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, using an explicit <wire/> element. 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://docs.oasis-open.org/ns/opencsa/sca/200712"
               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"/>
   </component>

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

   <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 the SCA runtime MUST search within the composite for target services which are compatible with the reference. [ASM60022] "Compatible" here means:

- the target service interface MUST be a compatible superset of the reference interface when using autowire to wire a reference (as defined in the section on Wires) [ASM60023]
- the intents, and policies applied to the service MUST be compatible with those on the reference when using autowire to wire a reference – so that wiring the reference to the service will not cause an error due to policy mismatch [ASM60024] (see the Policy Framework specification [10] for details)

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

- for an autowire reference with multiplicity 0..1 or 1..1, the SCA runtime MUST wire the reference to one of the set of valid target services chosen from the set in a runtime-dependent fashion [ASM60025]
- for an autowire reference with multiplicity 0..n or 1..n, the reference MUST be wired to all of the set of valid target services [ASM60026]

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

- for an autowire reference with multiplicity 0..1 or 0..n, if the SCA runtime finds no valid target service, there is no problem – no services are wired and the SCA runtime MUST NOT raise an error [ASM60027]
- for an autowire reference with multiplicity 1..1 or 1..n, if the SCA runtime finds no valid target services an error MUST be raised by the SCA runtime since the reference is intended to be wired [ASM60028]

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:

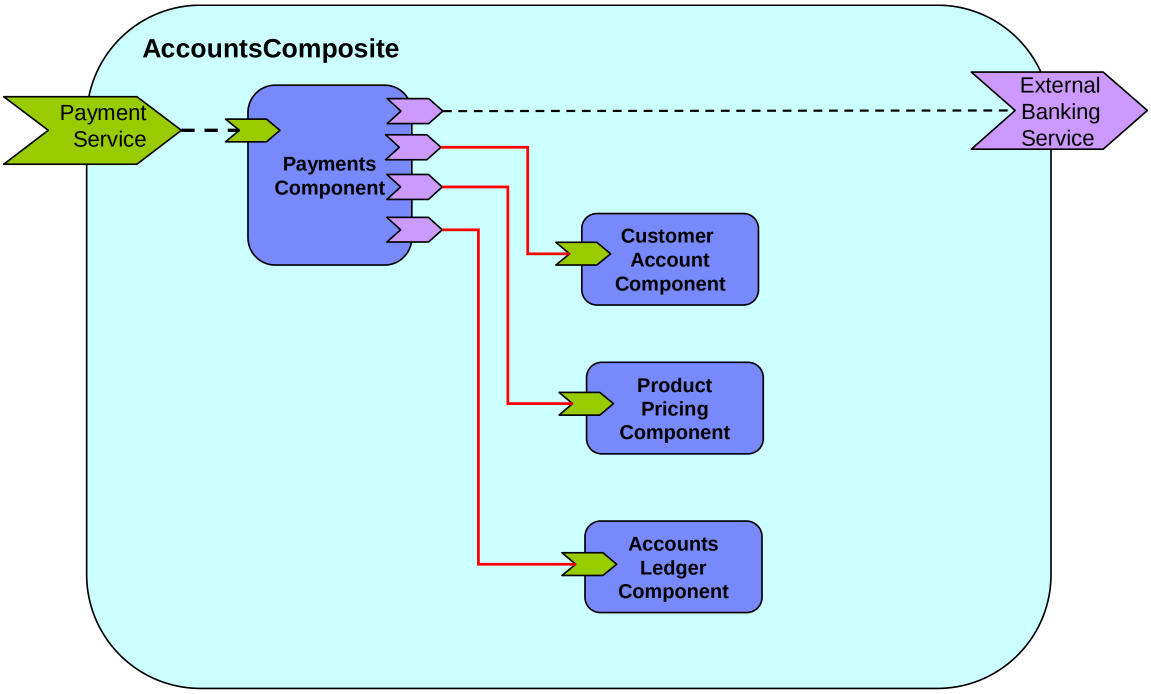

AccountsComposite
Payment Service
Payments Component
External Banking Service
Customer Account Component
Product Pricing Component
Accounts Ledger Component

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://docs.oasis-open.org/ns/opencsa/sca/200712"
    xmlns:foo="http://foo.com"
    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.java class="com.foo.accounts.ProductPricing"/>
    </component>

    <component name="AccountsLedgerComponent">
```

```
            <implementation.composite name="foo:AccountsLedgerComposite"/>
        </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://docs.oasis-open.org/ns/opencsa/sca/200712"
     xmlns:foo="http://foo.com"
    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.java class="com.foo.accounts.ProductPricing"/>
    </component>

    <component name="AccountsLedgerComponent">
        <implementation.composite name="foo:AccountsLedgerComposite"/>
    </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. The boundary of visibility, sometimes called encapsulation, can be enforced when assembling components and composites, but such encapsulation structures might not be enforceable in a particular implementation language.

A composite used as a component implementation must also honor a completeness contract. The services, references and properties of the composite form a contract (represented by the component type of the composite) which is relied upon by the using component. The concept of completeness of the composite implies that, once all <include/> element processing is performed on the composite:

1. For a composite used as a component implementation, each composite service offered by the composite MUST promote a component service of a component that is within the composite. [ASM60032]

2. For a composite used as a component implementation, every component reference of components within the composite with a multiplicity of 1..1 or 1..n MUST be wired or promoted (according to the various rules for specifying target services for a component reference described in section 5.3.1). [ASM60033]

3. For a composite used as a component implementation, all properties of components within the composite, where the underlying component implementation specifies "mustSupply=true" for the property, MUST either specify a value for the property or source the value from a composite property. [ASM60034]

The component type of a composite is defined by the set of composite service elements, composite reference elements and composite 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:

```
<!-- implementation.composite pseudo-schema -->
<implementation.composite name="xs:QName" requires="list of xs:QName"?
policySets="list of xs:QName"?>
```

The implementation.composite element has the following attributes:

- ***name (1..1)*** – the name of the composite used as an implementation. The @name attribute of an <implementation.composite/> element MUST contain the QName of a composite in the SCA Domain. [ASM60030]
- ***requires : QName (0..n)*** – a list of required policy intents. See the Policy Framework specification [10] for a description of this attribute. Specified ***required intents*** add to or further qualify the required intents defined for the promoted component reference.
- ***policySets : QName (0..n)*** – a list of policy sets. See the Policy Framework specification [10] for a description of this attribute.

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://docs.oasis-open.org/ns/opencsa/sca/200712
    file:/C:/Strategy/SCA/v09_osoaschemas/schemas/sca.xsd"
    xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
    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 may include another composite by using the **include** element. This provides a recursive inclusion capability. The semantics of included composites are that the element content children of the included composite are inlined, with certain modification, into the using composite. This is done recursively till the resulting composite does not contain an **include** element. The outer included composite element itself is discarded in this process – only its contents are included as described below:

1. All the element content children of the included composite are inlined in the including composite.
2. The attributes **targetNamespace**, **name**, **constrainingType**, and **local** of the included composites are discarded.
3. All the namespace declaration on the included composite element are added to the inlined element content children unless the namespace binding is overridden by the element content children.
4. The attribute **autowire**, if specified on the included composite, is included on all inlined component element children unless the component child already specifies that attribute.
5. The attribute values of **requires** and **policySet**, if specified on the included composite, are merged with corresponding attribute on the inlined component, service and reference children elements. Merge in this context means a set union.
6. Extension attributes ,if present on the included composite, must follow the rules defined for that extension. Authors of attribute extensions on the composite element must define rules for inclusion.

If the included composite has the value *true* for the attribute **local** then the including composite must have the same value for the **local** attribute, else it is considered an error.

The composite file used for inclusion can have any contents, but its document root element must be ***composite***. 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.

The SCA runtime MUST raise an error if the composite resulting from the inclusion of one composite into another is invalid. [ASM60031] For example, it is an error if there are duplicated elements in the using composite (eg. two services with the same uri contributed by different included composites). It is not considered an erorr if the (using) composite resulting from the inclusion is incomplete (eg. wires with non-existent source or target). Such incomplete resulting composites are permitted to allow recursive composition.

The following snippet shows the pseudo-schema for the include element.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Include snippet -->
<composite ...>
   ...
   <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.

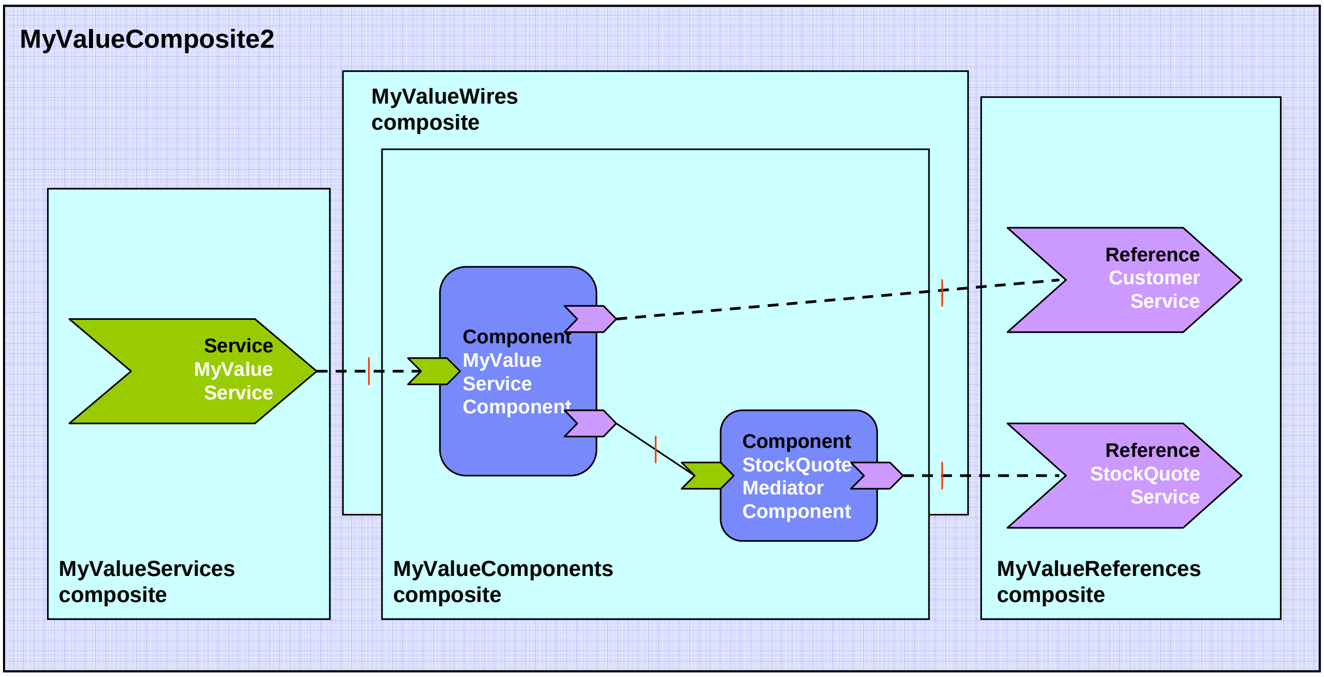

MyValueComposite2
MyValueWires composite
Service MyValue Service
Component MyValue Service Component
Component StockQuote Mediator Component
Reference Customer Service
Reference StockQuote Service
MyValueServices composite
MyValueComponents composite
MyValueReferences composite

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://docs.oasis-open.org/ns/opencsa/sca/200712"
               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://docs.oasis-open.org/ns/opencsa/sca/200712"
               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://docs.oasis-open.org/ns/opencsa/sca/200712"
               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://docs.oasis-open.org/ns/opencsa/sca/200712"
               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://docs.oasis-open.org/ns/opencsa/sca/200712"
               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 can 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 Structural URI of Components

The ***structural URI*** is a relative URI that describes each use of a given component in the Domain, relative to the URI of the domain itself. It is never specified explicitly, but it calculated from the configuration of the components configured into the Domain.

A component in a composite may be used more than once in the domain, if its containing composite is used as the implementation of more than one higher-level component. The structural URI may be used to separately identify each use of a component - for example, the structural URI may be used to attach different policies to each separate use of a component.

For components directly deployed into the domain, the structural URI is simply the name of the component.

Where components are nested within a composite which is used as the implementation of a higher level component, the structural URI consists of the name of the nested component prepended with each of the names of the components upto and including the domain level component.

For example, consider a component named Component1 at the domain level, where its implementation is Composite1 which in turn contains a component named Component2, which is implemented by Composite2 which contains a component named Component3. The three components in this example have the following structural URIs:

1. Component1:   Component1
2. Component2:   Component1/Component2
3. Component3:   Component1/Component2/Component3

The structural URI can also be extended to refer to specific parts of a component, such as a service or a reference, by appending an appropriate fragment identifier to the component's structural URI, as follows:

- Service:
  #service(servicename)

- Reference:
  #reference(referencename)

- Service binding:
  #service-binding(servicename/bindingname)

- Reference binding:
  #reference-binding(referencename/bindingname)

So, for example, the structural URI of the service named "testservice" of component "Component1" is Component1#service(testservice).

7 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 by the implementation of the component to which the constrainingType is attached. [ASM70001] 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 specified on the component element, the SCA runtime MUST raise an error. [ASM70002]

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://docs.oasis-
open.org/ns/opencsa/sca/200712"
               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 (1..1)*** – the name of the constrainingType. The form of a constraingType name is an XML QName, in the namespace identified by the targetNamespace attribute. The name attribute of the constraining type MUST be unique in the SCA domain. [ASM70003]
- ***targetNamespace (0..1) –*** an identifier for a target namespace into which the constrainingType is declared
- ***requires (0..1)*** – 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 its component type MUST contain all the services, references and properties specified in the constrainingType. [ASM70004] The constraining type's references and services will have interfaces specified and can have intents specified. An implementation MAY contain additional services, additional optional references (multiplicity 0..1 or 0..n) and additional optional properties beyond those declared in the constraining type, but MUST NOT contain additional non-optional references (multiplicity 1..1 or 1..n) or additional non-optional properties (a property with mustSupply=true). [ASM70005]

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. Additional services, references and properties provided by the implementation which are not declared in the constrainingType associated with a component MUST NOT be configured in any way by the containing composite. [ASM70006] 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. A component or implementation can use a qualified form of an intent specified in unqualified form in the constrainingType, but if the constrainingType uses the qualified form of an intent, then the component or implementation MUST also use the qualified form, otherwise there is an error. [ASM70007]

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.

7.1 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.

8 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 can be simple types such as a string value or they can be complex types.

SCA currently supports the following interface type systems:

- Java interfaces
- WSDL 1.1 portTypes (Web Services Definition Language [8])
- C++ classes
- Collections of 'C' functions

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 definition for the ***interface*** base element.

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

The ***interface*** base element has the following ***attributes***:

- ***requires : QName (0..n)*** – a list of policy intents. See the Policy Framework specification [10] for a description of this attribute
- ***policySets : QName (0..n)*** – a list of policy sets. See the Policy Framework specification [10] for a description of this attribute.

For information about Java interfaces, including details of SCA-specific annotations, see the SCA Java Common Annotations and APIs specification [SCA-Common-Java].

For information about WSDL interfaces, including details of SCA-specific extensions, see SCA-Specific Aspects for WSDL Interfaces and WSDL Interface Type.

For information about C++ interfaces, see the SCA C++ Client and Implementation Model specification [SCA-CPP-Client].

For information about C interfaces, see the SCA C Client and Implementation Model specification [SCA-C-Client].

8.1 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. WSDL defined interfaces are always remotable. See the relevant specifications for details of interfaces defined using other languages.

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***. [ASM80002] This restriction on operation overloading for remotable services aligns with the WSDL 2.0 specification, which disallows operation overloading, and also with the WS-I Basic Profile 1.1 (section 4.5.3 - R2304) which has a constraint which disallows operation overloading when using WSDL 1.1.

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 can modify input messages (parameters) during or after an invocation and can modify return messages (results) after the invocation. If a remotable service is called locally or remotely, the SCA container MUST ensure sure that no modification of input messages by the service or post-invocation modifications to return messages are seen by the caller. [ASM80003]

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***.

8.2 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 needs to 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 MUST provide an implementation of the callback interface. [ASM80004]

Callbacks can 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. [ASM80005]

Note that an interface document such as a WSDL file or a Java interface can contain annotations that declare a callback interface for a particular interface (see the section on WSDL Interface type and the Java Common Annotations and APIs specification [SCA-Common-Java]). Whenever an interface document declaring a callback interface is used in the declaration of an <interface/> element in SCA, it MUST be treated as being bidirectional with the declared callback interface. [ASM80010] In such cases, there is no requirement for the <interface/> element to declare the callback interface explicitly.

If an <interface/> element references an interface document which declares a callback interface and also itself contains a declaration of a callback interface, the two callback interfaces MUST be compatible. [ASM80011]

Where a component uses an implementation and the component configuration explicitly declares an interface for a service or a reference, if the matching service or reference declaration in the component type declares an interface which has a callback interface, then the component interface declaration MUST also declare a compatible interface with a compatible callback interface. [ASM80012] If the service or reference declaration in the component type declares an interface without a callback interface, then the component configuration for the corresponding service or reference MUST NOT declare an interface with a callback interface. [ASM80013]

Where a composite declares an interface for a composite service or a composite reference, if the promoted service or promoted reference has an interface which has a callback interface, then the interface declaration for the composite service or the composite reference MUST also declare a compatible interface with a compatible callback interface. [ASM80014] If the promoted service or promoted reference has an interface without a callback interface, then the interface declaration for the composite service or composite reference MUST NOT declare a callback interface. [ASM80015]

See Section 6.4 Wires for a definition of "compatible interfaces".

In a bidirectional interface, the service interface can have more than one operation defined, and the callback interface can also have more than one operation defined. SCA runtimes MUST allow an invocation of any operation on the service interface to be followed by zero, one or many invocations of any of the operations on the callback interface. [ASM80009] These callback operations can be invoked either before or after the operation on the service interface has returned a response message, if there is one.

For a given invocation of a service operation, which operations are invoked on the callback interface, when these are invoked, the number of operations invoked, and their sequence are not described by SCA. It is possible that this metadata about the bidirectional interface can be supplied through mechanisms outside SCA. For example, it might be provided as a written description attached to the callback interface.

8.3 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.

There is a policy intent with the name ***conversational*** which is used to mark an interface as being conversational in nature. Where a service or a reference has a conversational interface, the conversational intent MUST be attached either to the interface itself, or to the service or reference using the interface. [ASM80006] How to attach the conversational intent to an interface depends on the type of the interface. For a WSDL interface, this is described in section 8.4 "SCA-Specific Aspects for WSDL Interfaces". For a Java interface, it is described in the Java Common Annotations and APIs specification. Note that setting the conversational intent on the service or reference element is useful when reusing an existing interface definition that contains no SCA information, since it requires no modification of the interface artifact.

The meaning of the conversational intent is that both the client and the provider of the interface can 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 can start a conversation. The decision on whether an operation starts a conversation depends on the component's implementation and its implementation type. Implementation types can support components which provide conversational services. If an implementation type does provide this support, the specification for that implementation type defines 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 can be annotated with an ***endsConversation*** annotation (the mechanism for annotating the interface depends on the interface type) which indicates that when the operation is invoked, the conversation is at an end. Where an interface is ***bidirectional***, operations may also be annotated in this way on operations of the callback interface. When a conversation ending operation is called, it indicates to both the client and the service provider that the conversation is complete. Once an operation marked with endsConversation has been invoked, any subsequent attempts to call an operation or a callback operation associated with the same conversation MUST generate a sca:ConversationViolation fault. [ASM80007]

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://docs.oasis-open.org/ns/opencsa/sca/200712.`

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 can 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 can have an API to access the ID associated with the conversation, although no assumptions can be made about the structure of that identifier. Implementation types can also have 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 could also manage the conversational state manually.

8.4 Long-running Request-Response Operations

8.4.1 Background

A service offering one or more operations which map to a WSDL request-response pattern may be implemented in a long-running, potentially interruptible, way. Consider a BPEL process with receive and reply activities referencing the WSDL request-response operation. Between the two activities, the business process logic may be a long-running sequence of steps, including activities causing the process to be interrupted. Typical examples are steps where the process waits for another message to arrive or a specified time interval to expire, or the process may perform asynchronous interactions such as service invocations bound to asynchronous protocols or user interactions. This is a common situation in business processes, and it causes the implementation of the WSDL request-response operation to run for a very long time, e.g., several months (!). In this case, it is not meaningful for any caller to remain in a synchronous wait for the response while blocking system resources or holding database locks.

Note that it is possible to model long-running interactions as a pair of two independent operations as described in the section on bidirectional interfaces. However, it is a common practice (and in fact much more convenient) to model a request-response operation and let the infrastructure deal with the asynchronous message delivery and correlation aspects instead of putting this burden on the application developer.

8.4.2 Definition of "long-running"

A request-response operation is considered long-running if the implementation does not guarantee the delivery of the response within any specified time interval. Clients invoking such request-response operations are strongly discouraged from making assumptions about when the response can be expected.

8.4.3 The asyncInvocation Intent

This specification permits a long-running request-response operation or a complete interface containing such operations to be marked using a policy intent with the name ***asyncInvocation***. It is also possible for a service to set the asyncInvocation. intent when using an interface which is not marked with the asyncInvocation. intent. This can be useful when reusing an existing interface definition that does not contain SCA information.

8.4.4 Requirements on Bindings

In order to support a service operation which is marked with the asyncInvocation intent, it is necessary for the binding (and its associated policies) to support separate handling of the request message and the response message. Bindings which only support a synchronous style of message handling, such as a conventional HTTP binding, cannot be used to support long-running operations.

The requirements on a binding to support the asyncInvocation intent are the same as those required to support services with bidirectional interfaces - namely that the binding needs to be able to treat the transmission of the request message separately from the transmission of the response message, with an arbitrarily large time interval between the two transmissions.

An example of a binding/policy combination that supports long-running request-response operations is a Web service binding used in conjunction with the WS-Addressing "wsam:NonAnonymousResponses" assertion.

8.4.5 Implementation Type Support

SCA implementation types can provide special asynchronous client-side and asynchronous server-side mappings to assist in the development of services and clients for long-running request-response operations.

8.5 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://docs.oasis-open.org/ns/opencsa/sca/200712) needs to 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). The attribute contains one or more intent names, as defined by the Policy Framework specification [10]. Any service or reference that uses an interface marked with required intents MUST implicitly add those intents to its own @requires list. [ASM80008]

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>
   ...
```

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>
   ...
```

SCA defines an attribute which is used to indicate that a given WSDL Port Type element (WSDL 1.1) has an associated callback interface. This is the @callback attribute, which applies to a WSDL <portType/> element.

The @callback attribute is defined as a global attribute in the SCA namespace, as follows:

```
   <attribute name="callback" type="QName"/>
```

The value of the @callback attribute is the QName of a Port Type. The port type declared by the @callback attribute is the callback interface to use for the portType which is annotated by the @callback attribute.

Here is an example of a portType element with a callback attribute:

```

   <portType name="LoanService" sca:callback="foo:LoanServiceCallback">
         <operation name="apply">
               <input message="tns:ApplicationInput"/>
               <output message="tns:ApplicationOutput"/>
         </operation>
         ...
   </portType>
```

8.6 WSDL Interface Type

The WSDL interface type is used to declare interfaces for services and for references, where the interface is defined in terms of a WSDL document. An interface is defined in terms of a WSDL 1.1 Port Type with the arguments and return of the service operations described using XML schema.

A WSDL interface is declared by an ***interface.wsdl*** element. The following shows the pseudo-schema for the interface.wsdl element:

```
<!-- WSDL Interface schema snippet -->
<interface.wsdl interface="xs:anyURI" callbackInterface="xs:anyURI"?>
```

The interface.wsdl element has the following ***attributes***:

- ***interface (1..1)*** - the URI of a WSDL Port Type

The interface.wsdl @interface attribute MUST reference a portType of a WSDL 1.1 document. [ASM80001]

- ***callbackInterface(0..1)*** - an optional callback interface, which is the URI of a WSDL Port Type

  The interface.wsdl @callbackInterface attribute, if present, MUST reference a portType of a WSDL 1.1 document. [ASM80016]

The form of the URI for WSDL port types follows the syntax described in the WSDL 1.1 Element Identifiers specification [WSDL11_Identifiers]

8.6.1 Example of interface.wsdl

```
<interface.wsdl interface="http://www.stockquote.org/StockQuoteService#
                           wsdl.porttype(StockQuote)"
callbackInterface="http://www.stockquote.org/StockQuoteService#
                        wsdl.porttype(StockQuoteCallback)"/>
```

This declares an interface in terms of the WSDL port type "StockQuote" with a callback interface defined by the "StockQuoteCallback" port type.

9 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 ... >
   ...
   <service ... >*
      <interface … />?
      <binding uri="xs:anyURI"? name="xs:NCName"?
         requires="list of xs:QName"?
         policySets="list of xs:QName"?>*
         <operation name="xs:NCName" requires="list of xs:QName"?
            policySets="list of xs:QName"?/>*
         <wireFormat/>?
         <operationSelector/>?
      </binding>
      <callback>?
         <binding uri="xs:anyURI"? name="xs:NCName"?
            requires="list of xs:QName"?
            policySets="list of xs:QName"?>+
            <operation name="xs:NCName" requires="list of xs:QName"?
               policySets="list of xs:QName"?/>*
            <wireFormat/>?
            <operationSelector/>?
         </binding>
      </callback>
   </service>
   ...
   <reference ... >*
      <interface … />?
      <binding uri="xs:anyURI"? name="xs:NCName"?
         requires="list of xs:QName"?
         policySets="list of xs:QName"?>*
         <operation name="xs:NCName" requires="list of xs:QName"?
            policySets="list of xs:QName"?/>*
         <wireFormat/>?
         <operationSelector/>?
      </binding>
      <callback>?
         <binding uri="xs:anyURI"? name="xs:NCName"?
            requires="list of xs:QName"?
            policySets="list of xs:QName"?>+
            <operation name="xs:NCName" requires="list of xs:QName"?
               policySets="list of xs:QName"?/>*
            <wireFormat/>?
            <operationSelector/>?
         </binding>
      </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 (0..1) -*** has the following semantic.
    - The uri attribute can be omitted.
    - For a binding of a ***reference*** the URI attribute defines the target URI of the reference. This MUST be either the componentName/serviceName for a wire to an endpoint within the SCA domain, or the accessible address of some service endpoint either inside or outside the SCA domain (where the addressing scheme is defined by the type of the binding). [ASM90001]
    - The circumstances under which the uri attribute can be used are defined in section "5.3.1 Specifying the Target Service(s) for a Reference."
    - 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 value of the name attribute of the binding.
- ***name (0..1)*** – a name for the binding instance (an NCName). 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 binding can have the default name value; all others must have a name value specified that is unique within the service or reference. [ASM90002] 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 (0..1)*** - a list of policy intents. See the Policy Framework specification [10] for a description of this attribute.
- ***policySets (0..1)*** – a list of policy sets. See the Policy Framework specification [10] for a description of this attribute.

A binding element has the following child elements:

- ***operation: Operation (0..n)*** - Zero or more operation elements. These elements are used to describe characteristics of individual operations within the interface. For a detailed decription of the operation element, see the Policy Framework specification [SCA Policy].
- ***wireFormat (0..1)*** - a wireFormat to apply to the data flowing using the binding. See the wireFormat section for details.
- ***operationSelector(0..1)*** - an operationSelector element that is used to match a particular message to a particular operation in the interface. See the operationSelector section for details

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 needs to be documented by the runtime.

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

If a reference has any bindings they MUST be resolved which means that each binding MUST include a value for the @URI attribute or MUST otherwise specify an endpoint. The reference MUST NOT be wired using other SCA mechanisms. [ASM90003] To specify constraints on the kinds of bindings that are acceptable for use with a reference, the user specifies either policy intents or policy sets.

Users can also specifically wire, not just to a component service, but to a specific binding offered by that target service. To do so, a wire target MAY be specified with a syntax of "componentName/serviceName/bindingName". [ASM90004]

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

9.1 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 can make this information available to component implementations in their execution context. The specifications for these implementation types describe how this information is accessed and in what form it is presented.

9.2 WireFormat

A wireFormat is the form that a data structure takes when it is transmitted using some communication binding. Another way to describe this is "the form that the data takes on the wire". A wireFormat can be specific to a given communication method, or it may be general, applying to many different communication methods. An example of a general wireFormat is XML text format.

Where a particular SCA binding can accommodate transmitting data in more than one format, the configuration of the binding MAY include a definition of the wireFormat to use. This is done using an optional <sca:wireFormat/> subelement of the <binding/> element.

Where a binding supports more than one wireFormat, the binding defines one of the wireFormats to be the default wireFormat which applies if no <wireFormat/> subelement is present.

The base sca:wireFormat element is abstract and it has no attributes and no child elements. For a particular wireFormat, an extension subtype is defined, using substitution groups, for example:

- <sca:wireFormat.xml/>
- A wireFormat that transmits the data as an XML text datastructure
- <sca:wireFormat.jms/>
- The "default JMS wireFormat" as described in the JMS Binding specification

Specific wireFormats can have elements that include either attributes or subelements or both.

For details about specific wireFormats, see the related SCA Binding specifications.

9.3 OperationSelector

An operationSelector is necessary for some types of transport binding where messages are transmitted across the transport without any explicit relationship between the message and the interface operation to which it relates. SOAP is an example of a protocol where the messages do contain explicit information that relates each message to the operation it targets. However, other transport bindings have messages where this relationship is not expressed in the message or in any related headers (pure JMS messages, for example). In cases where the messages arrive at a service without any explicit information that maps them to specific operations, it is necessary for the metadata attached to the service binding to contain the required mapping information. The information is held in an operationSelector element which is a child element of the binding element.

The base sca:operationSelector element is abstract and it has no attributes and no child elements. For a particular operationSelector, an extension subtype is defined, using substitution groups, for example:

- <sca:operationSelector.XPath/>
- An operation selector that uses XPath to filter out specific messages and target them to particular named operations.

Specific operationSelectors can have elements that include either attributes or subelements or both.

For details about specific operationSelectors, see the related SCA Binding specifications.

9.4 Form of the URI of a Deployed Binding

SCA Bindings specifications can choose to use the ***structural URI*** defined in the section "Structural URI of Components" above to derive a binding specific URI according to some Binding-related scheme. The relevant binding specification describes this.

Alternatively, <binding/> elements have an optional @URI attribute, which is termed a bindingURI.

If the bindingURI is specified on a given <binding/> element, the binding can optionally use it to derive an endpoint URI relevant to the binding. The derivation is binding specific and is described by the relevant binding specification.

For binding.sca, which is described in the SCA Assembly specification, this is as follows:

- If the binding uri attribute is specified on a reference, it identifies the target service in the SCA domain by specifying the service's structural URI.
- If the binding uri attribute is specified on a service, it is ignored.

9.4.1 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 needs to offer a different mechanism for specifying the service address.

9.4.2 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".

9.5 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 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 definition and the SCA binding should be one of them.

If a reference does not have a binding, then the binding used can be any of the bindings specified by the service provider, as long as the intents required by the reference and the service are all respected.

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>

9.5.1 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://docs.oasis-open.org/ns/opencsa/sca/200712"
               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>
```

9.6 Web Service Binding

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

9.7 JMS Binding

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

10 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 SCA contributions in files called META-INF/definitions.xml (relative to the contribution base URI). Although the definitions are specified within a single SCA contribution, the definitions are visible throughout the domain. Because of this, all of the QNames for the definitions contained in definitions.xml files MUST be unique within the domain. [ASM10001] 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://docs.oasis-open.org/ns/opencsa/sca/200712"
               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].

11 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], [9], [11]) 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 are all part of the SCA namespace ("http://docs.oasis-open.org/ns/opencsa/sca/200712"), 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 are 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.

11.1 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://docs.oasis-open.org/ns/opencsa/sca/200712"
        xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
        elementFormDefault="qualified">

   ...

    <element name="interface" type="sca:Interface" abstract="true"/>
    <complexType name="Interface"/>
    <complexType name="Interface" abstract="true">
     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
    </complexType>


   ...

</schema>
```

In the following snippet is an example of 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://docs.oasis-open.org/ns/opencsa/sca/200712"
        xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712">

   <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 is 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://docs.oasis-open.org/ns/opencsa/sca/200712"
        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>
```

11.2 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://docs.oasis-open.org/ns/opencsa/sca/200712"
        xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
        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://docs.oasis-open.org/ns/opencsa/sca/200712"
        xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712">

   <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 is 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://docs.oasis-open.org/ns/opencsa/sca/200712"
        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 (1..1)*** – 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 (0..1)*** – a set of intents which the implementation type always provides. See the Policy Framework specification [10] for details.
- ***mayProvide (0..1)*** – a set of intents which the implementation type may provide. See the Policy Framework specification [10] for details.

11.3 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://docs.oasis-open.org/ns/opencsa/sca/200712"
        xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
        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 is an example of 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://docs.oasis-open.org/ns/opencsa/sca/200712"
        xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712">

   <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 is 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://docs.oasis-open.org/ns/opencsa/sca/200712"
              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 (1..1)*** – 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 (0..1)*** – a set of intents which the binding type always provides. See the Policy Framework specification [10] for details.
- ***mayProvide (0..1)*** – a set of intents which the binding type may provide. See the Policy Framework specification [10] for details.

11.4 Defining an Import Type

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

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright(C) OASIS(R) 2005,2008. All Rights Reserved. OASIS trademark,
IPR and other policies apply.  -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
   targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
   elementFormDefault="qualified">

...

   <!-- Import -->
   <element name="importBase" type="sca:Import" abstract="true" />
   <complexType name="Import" abstract="true">
      <complexContent>
         <extension base="sca:CommonExtensionBase">
            <sequence>
               <any namespace="##other" processContents="lax" minOccurs="0"
                  maxOccurs="unbounded"/>
            </sequence>
         </extension>
      </complexContent>
   </complexType>

   <element name="import" type="sca:ImportType"
      substitutionGroup="sca:importBase"/>
   <complexType name="ImportType">
      <complexContent>
         <extension base="sca:Import">
            <attribute name="namespace" type="string" use="required"/>
            <attribute name="location" type="anyURI" use="required"/>
         </extension>
      </complexContent>
   </complexType>

...

</schema>
```

In the following snippet we show how the base import definition is extended to support Java imports. In the import element, the namespace is expected to be an XML namespace, an import.java element uses a Java package name instead. The snippet shows the definition of the ***import.java*** element and the ***JavaImportType*** type contained in ***sca-import-java.xsd***.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
        xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712">

   <element name="import.java" type="sca:JavaImportType"
      substitutionGroup="sca:importBase"/>
   <complexType name="JavaImportType">
      <complexContent>
         <extension base="sca:Import">
            <attribute name="package" type="xs:String" use="required"/>
            <attribute name="location" type="xs:AnyURI" use="optional"/>
         </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-import-extension*** element and the ***my-import-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://docs.oasis-open.org/ns/opencsa/sca/200712"
        xmlns:tns="http://www.example.org/myextension">

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

For a complete example using this extension point, see the definition of ***import.java*** in the SCA Java Common Annotations and APIs Specification [SCA-Java].

11.5 Defining an Export Type

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

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright(C) OASIS(R) 2005,2008. All Rights Reserved. OASIS trademark,
IPR and other policies apply.  -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
   targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
   elementFormDefault="qualified">

...
   <!-- Export -->
   <element name="exportBase" type="sca:Export" abstract="true" />
   <complexType name="Export" abstract="true">
      <complexContent>
         <extension base="sca:CommonExtensionBase">
            <sequence>
               <any namespace="##other" processContents="lax" minOccurs="0"
                  maxOccurs="unbounded"/>
            </sequence>
         </extension>
      </complexContent>
   </complexType>

   <element name="export" type="sca:ExportType"
      substitutionGroup="sca:exportBase"/>
   <complexType name="ExportType">
      <complexContent>
         <extension base="sca:Export">
            <attribute name="namespace" type="string" use="required"/>
         </extension>
      </complexContent>
   </complexType>
...
</schema>
```

The following snippet shows how the base definition is extended to support Java exports. In a base *export* element, the *@namespace* attribute specifies XML namespace being exported. An *export.java* element uses a *@package* attribute to specify the Java package to be exported. The snippet shows the definition of the ***export.java*** element and the ***JavaExport*** type contained in ***sca-export-java.xsd***.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
        xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712">

   <element name="export.java" type="sca:JavaExportType"
      substitutionGroup="sca:exportBase"/>
   <complexType name="JavaExportType">
      <complexContent>
         <extension base="sca:Export">
            <attribute name="package" type="xs:String" 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-export-extension*** element and the ***my-export-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:// docs.oasis-open.org/ns/opencsa/sca/200712"
        xmlns:tns="http://www.example.org/myextension">

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

For a complete example using this extension point, see the definition of ***export.java*** in the SCA Java Common Annotations and APIs Specification [SCA-Java].

12 Packaging and Deployment

12.1 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.

12.2 Contributions

An SCA domain might 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:

- For any contribution 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 [ASM12001]

- Within any contribution packaging A directory resource SHOULD exist at the root of the hierarchy named META-INF [ASM12002]
- Within any contribution packaging 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. [ASM12003]

  The same document also optionally lists namespaces of constructs that are defined within the contribution and which may be used by other contributions
  Optionally, in the sca-contribution.xml file, additional elements MAY exist that list the namespaces of constructs that are needed by the contribution and which are be found elsewhere, for example in other contributions. [ASM12004] 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.

12.2.1 SCA Artifact Resolution

Contributions can 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 can 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 can be to SCA artifacts such as composites or they can be to other artifacts such as WSDL files, XSD files or to code artifacts such as Java class files and BPEL process files. Note: This form of artifact resolution does not apply to imports of composite files, as described in Section 6.6.

A contribution can 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 by the SCA runtime to resolve artifact dependencies. [ASM12005] The SCA runtime MUST raise an error if an artifact cannot be resolved using these mechanisms, if present. [ASM12021]

SCA also provides an artifact resolution mechanism. The SCA artifact resolution mechanism is used either where no other mechanisms are available, for example 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 relates to a mainframe COBOL service whose interfaces are declared using a 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. SCA artifact resolution is a general mechanism that can be extended for the handling of specific types of artifact. The general mechanism that is described in the following paragraphs is mainly intended for the handling of XML artifacts. Other types of artifacts, for example Java classes, use an extended version of artifact resolution that is specialized to their nature (eg. instead of "namespaces", Java uses "packages"). Descriptions of these more specialized forms of artifact resolution are contained in the SCA specifications that deal with those artifact types.

Import and export statements for XML artifacts work at the level of namespaces - so that an import statement declares that artifacts from a specified namespace are found in other contributions, while an export statement makes all the artifacts from a specified namespace available to other contributions.

An import declaration can simply specify the namespace to import. In this case, the locations which are searched for artifacts in that namespace are the contribution(s) in the Domain which have export declarations for the same namespace, if any. Alternatively an import declaration can specify a location from which artifacts for the namespace are obtained, in which case, that specific location is searched. There can be multiple import declarations for a given namespace. Where multiple import declarations are made for the same namespace, all the locations specified MUST be searched in lexical order. [ASM12022]

For an XML namespace, artifacts can be declared in multiple locations - for example a given namespace can have a WSDL declared in one contribution and have an XSD defining XML data types in a second contribution.

If the same artifact is declared in multiple locations, this is not an error. The first location as defined by lexical order is chosen. If no locations are specified no order exists and the one chosen is implementation dependent.

When a contribution contains a reference to an artifact from a namespace that is declared in an import statement of the contribution, if the SCA artifact resolution mechanism is used to resolve the artifact, the SCA runtime MUST resolve artifacts in the following order:

1. from the locations identified by the import statement(s) for the namespace. Locations MUST NOT be searched recursively in order to locate artifacts (ie only a one-level search is performed).

2. from the contents of the contribution itself. [ASM12023]

When a contribution uses an artifact contained in another contribution through SCA artifact resolution, if that artifact itself has dependencies on other artifacts, the SCA runtime MUST resolve these dependencies in the context of the contribution containing the artifact, not in the context of the original contribution. [ASM12024]

For example:

- a first contribution "C1" references an artifact "A1" in the namespace "n1" and imports the "n1" namespace from a second contribution "C2".

- in contribution "C2" the artifact "A1" in the "n1" namespace references an artifact "A2" also in the "n1" namespace", which is resolved through an import of the "n1" namespace in "C2" which specifies the location "C3".

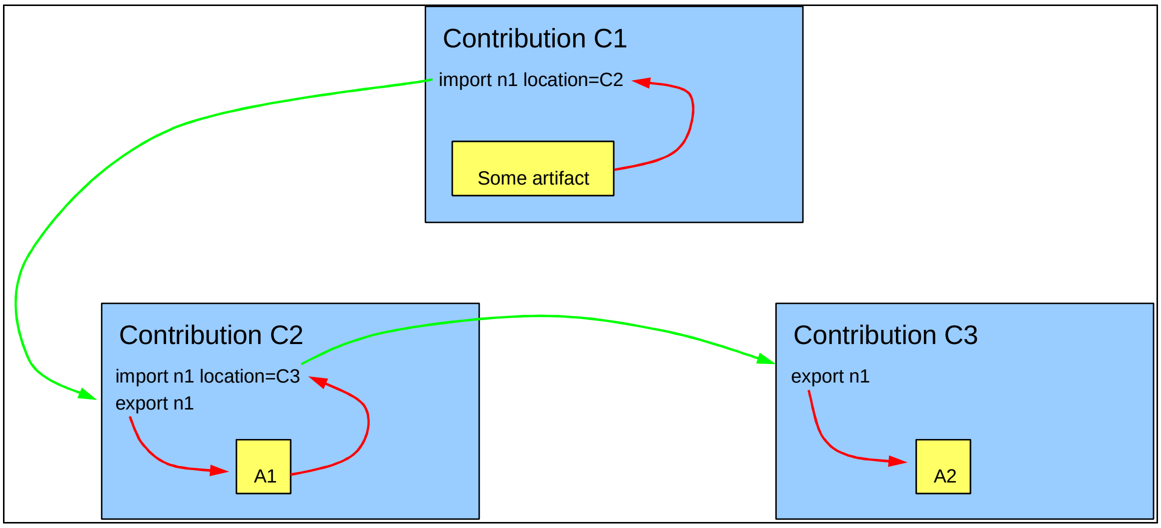


Figure 14: Example of SCA Artifact Resolution between Contributions

The "A2" artifact is contained within the third contribution "C3" from which it is resolved by the contribution "C2". The "C3" contribution is never used to resolve artifacts directly for the "C1" contribution, since "C3" is not declared as an import location for "C1".

For example, if for a contribution "C1",an import is used to resolve a composite "X1" contained in contribution "C2", and composite "X1" contains references to other artifacts such as WSDL files or XSDs, those references in "X1" are resolved in the context of contribution "C2" and not in the context of contribution "C1".

The SCA runtime MUST ignore local definitions of an artifact if the artifact is found through resolving an import statement. [ASM12024]

The SCA runtime MUST raise an error if an artifact cannot be resolved by the precedence order above. [ASM12025]

12.2.2 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 needs 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://docs.oasis-open.org/ns/opencsa/sca/200712>
```

```
    <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.

Attributes of the deployable element:

- ***composite (1..1)*** – 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.

Attributes of theexport element:

- ***namespace (1..1)*** – 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 can 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 is 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.

Attributes of the import element:

- ***namespace (1..1)*** – For XML definitions, which are identified by QNames, the namespace is 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 can 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 is a fully qualified package name.

- ***location (0..1)*** – 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 can point to another contribution (through its URI) or it can point to some location entirely outside the SCA Domain.

It is expected that SCA runtimes can 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 are expected to do so similarly when used as @schemaLocation and @wsdlLocation and other artifact location specifications.

The order in which the import statements are specified can 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 can 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 can 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.

12.2.3 Contribution Packaging using ZIP

SCA allows many different packaging formats that SCA runtimes can support, but SCA requires that all runtimes MUST support the ZIP packaging format for contributions. [ASM12006] This format allows that metadata specified by the section 'SCA Contribution Metadata Document' be present. Specifically, it can contain a top-level "META-INF" directory and a "META-INF/sca-contribution.xml" file and there can 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 can 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].

12.3 Installed Contribution

As noted in the section above, the contents of a contribution do 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 might or might 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 can 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 exported from multiple contributions. [ASM12007]

12.3.1 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).

12.4 Operations for Contributions

SCA Domains provide the following conceptual functionality associated with contributions (meaning the function might not be represented as addressable services and also meaning that equivalent functionality might be provided in other ways). The functionality is optional meaning that some SCA runtimes MAY choose not to provide the contribution functions functionality in any way. [ASM12008]

12.4.1 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 (<export/> elements) 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. [ASM12009]

Note that in many cases, the dependent contribution list can be generated. In particular, if the creator of 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.

12.4.2 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 might include only implementation code (e.g. PHP scripts). It is then 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.

12.4.3 remove Contribution

Removes the deployed contribution identified by a supplied contribution URI.

12.5 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, non-SCA artifact resolution mechanisms MUST be used by the SCA runtime in precendence to the SCA mechanisms. [ASM12010] 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 the non-SCA artifact resolution 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. [ASM12011]

12.6 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 value of @autowire for the logical domain composite MUST be autowire="false". [ASM12012]

For components at the Domain level, with References for which @autowire="true" applies, the behaviour of the SCA runtime for a given Domain MUST take ONE of the 3 following forms:

1) The SCA runtime MAY disallow deployment of any components with autowire References. In this case, the SCA runtime MUST generate an exception at the point where the component is deployed.

2) The SCA runtime MAY evaluate the target(s) for the reference at the time that the component is deployed and not update those targets when later deployment actions occur.

3) The SCA runtime MAY re-evaluate the target(s) for the reference dynamically as later deployment actions occur resulting in updated reference targets which match the new Domain configuration. How the new configuration of the reference takes place is described by the relevant client and implementation specifications.

[ASM12013]

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:

12.6.1 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).

12.6.2 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.

12.6.3 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.

12.6.4 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.

12.7 Dynamic Behaviour of Wires in the SCA Domain

For components with references which are at the Domain level, there is the potential for dynamic behaviour when the wires for a component reference change (this can only apply to component references at the Domain level and not to components within composites used as implementations):

The configuration of the wires for a component reference of a component at the Domain level can change by means of deployment actions:

1. <wire/> elements can be added, removed or replaced by deployment actions
2. Components can be updated by deployment actions (ie this may change the component reference configuration)
3. Components which are the targets of reference wires can be updated or removed
4. Components can be added that are potential targets for references which are marked with @autowire=true

Where <wire/> elements are added, removed or replaced by deployment actions, the components whose references are affected by those deployment actions MAY have their references updated by the SCA runtime dynamically without the need to stop and start those components. [ASM12014]

Where components are updated by deployment actions (their configuration is changed in some way, which may include changing the wires of component references), the new configuration MUST apply to all new instances of those components once the update is complete. [ASM12015] An SCA runtime MAY choose to maintain existing instances with the old configuration of components updated by deployment actions, but an SCA runtime MAY choose to stop and discard existing instances of those components. [ASM12016]

Where a component that is the target of a wire is removed, without the wire being changed, then future invocations of the reference that use that wire SHOULD fail with a ServiceUnavailable fault. If the wire is the result of the autowire process, the SCA runtime MUST:

- either cause future invocation of the target component's services to fail with a ServiceUnavailable fault
- or alternatively, if an alternative target component is available that satisfies the autowire process, update the reference of the source component [ASM12017]

Where a component that is the target of a wire is updated, future invocations of that reference SHOULD use the updated component. [ASM12018] Where an existing domain level component is updated, an SCA runtime MAY maintain a copy of a component offering a conversational service until all existing conversations complete - alternatively all existing conversations MAY be terminated. [ASM12019]

Where a component is added to the domain that is a potential target for a domain level component reference where that reference is marked as @autowire=true, the SCA runtime MUST:

- either update the references for the source component once the new component is running.
- or alternatively, defer the updating of the references of the source component until the source component is stopped and restarted. [ASM12020]

12.8 Dynamic Behaviour of Component Property Values

For a domain level component with a Property whose value is obtained from a Domain-level Property through the use of the @source attribute, if the domain level property is updated by means of deployment actions, the SCA runtime MUST

- either update the property value of the domain level component. once the update of the domain property is complete
- or alternative defer the updating of the component property value until the compoennt is stopped and restarted

13 Conformance

The XML schema available at the namespace URI, defined by this specification, is considered to be authoritative and takes precedence over the XML Schema defined in the appendix of this document.

An SCA runtime MUST reject a composite file that does not conform to the sca-core.xsd schema.. [ASM13001]

A. XML Schemas

A.1 sca.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright(C) OASIS(R) 2005,2008. All Rights Reserved. OASIS trademark,
IPR and other policies apply.  -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
   xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712">

   <include schemaLocation="sca-core-1.1-schema-200803.xsd"/>

   <include schemaLocation="sca-interface-java-1.1-schema-200803.xsd"/>
   <include schemaLocation="sca-interface-wsdl-1.1-schema-200803.xsd"/>
   <include schemaLocation="sca-interface-cpp-1.1-schema-200803.xsd"/>
   <include schemaLocation="sca-interface-c-1.1-schema-200803.xsd"/>

   <include schemaLocation="sca-implementation-java-1.1-schema-200803.xsd"/>
   <include schemaLocation=
            "sca-implementation-composite-1.1-schema-200803.xsd"/>
   <include schemaLocation="sca-implementation-cpp-1.1-schema-200803.xsd"/>
   <include schemaLocation="sca-implementation-c-1.1-schema-200803.xsd"/>
   <include schemaLocation="sca-implementation-bpel-1.1-schema-200803.xsd"/>

   <include schemaLocation="sca-binding-webservice-1.1-schema-200803.xsd"/>
   <include schemaLocation="sca-binding-jms-1.1-schema-200803.xsd"/>
   <include schemaLocation="sca-binding-sca-1.1-schema-200803.xsd"/>

   <include schemaLocation="sca-definitions-1.1-schema-200803.xsd"/>
   <include schemaLocation="sca-policy-1.1-schema-200803.xsd"/>

   <include schemaLocation="sca-contribution-1.1-schema-200803.xsd"/>

</schema>
```

A.2 sca-core.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright(C) OASIS(R) 2005,2008. All Rights Reserved. OASIS trademark,
IPR and other policies apply.  -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
   targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
   elementFormDefault="qualified">

   <import namespace="http://www.w3.org/XML/1998/namespace"
           schemaLocation="http://www.w3.org/2001/xml.xsd"/>

   <!-- Common extension base for SCA definitions -->
   <complexType name="CommonExtensionBase">
      <sequence>
```

```
         <element ref="sca:documentation" minOccurs="0"
            maxOccurs="unbounded"/>
      </sequence>
      <anyAttribute namespace="##other" processContents="lax"/>
   </complexType>

   <element name="documentation" type="sca:Documentation"/>
   <complexType name="Documentation" mixed="true">
      <sequence>
         <any namespace="##other" processContents="lax" minOccurs="0"
            maxOccurs="unbounded"/>
      </sequence>
      <attribute ref="xml:lang"/>
   </complexType>

   <!-- Component Type -->
   <element name="componentType" type="sca:ComponentType"/>
   <complexType name="ComponentType">
      <complexContent>
         <extension base="sca:CommonExtensionBase">
            <sequence>
               <element ref="sca:implementation" minOccurs="0"/>
               <choice minOccurs="0" maxOccurs="unbounded">
                  <element name="service" type="sca:ComponentService"/>
                  <element name="reference"
                     type="sca:ComponentTypeReference"/>
                  <element name="property" type="sca:Property"/>
               </choice>
               <any namespace="##other" processContents="lax" minOccurs="0"
                  maxOccurs="unbounded"/>
            </sequence>
            <attribute name="constrainingType" type="QName" use="optional"/>
         </extension>
      </complexContent>
   </complexType>

   <!-- Composite -->
   <element name="composite" type="sca:Composite"/>
   <complexType name="Composite">
      <complexContent>
         <extension base="sca:CommonExtensionBase">
            <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"/>
         </extension>
      </complexContent>
   </complexType>

   <!-- Contract base type for Service, Reference -->
   <complexType name="Contract" abstract="true">
      <complexContent>
         <extension base="sca:CommonExtensionBase">
            <sequence>
               <element ref="sca:interface" minOccurs="0" maxOccurs="1" />
               <element name="operation" type="sca:Operation" minOccurs="0"
                  maxOccurs="unbounded" />
               <element ref="sca:binding" minOccurs="0"
                  maxOccurs="unbounded"/>
               <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"/>
         </extension>
      </complexContent>
   </complexType>

   <!-- Service -->
   <complexType name="Service">
      <complexContent>
         <extension base="sca:Contract">
            <attribute name="promote" type="anyURI" use="required"/>
         </extension>
      </complexContent>
   </complexType>

   <!-- Interface -->
   <element name="interface" type="sca:Interface" abstract="true"/>
   <complexType name="Interface" abstract="true">
      <complexContent>
         <extension base="sca:CommonExtensionBase"/>
      </complexContent>
   </complexType>

   <!-- Reference -->
   <complexType name="Reference">
      <complexContent>
         <extension base="sca:Contract">
            <attribute name="autowire" type="boolean" use="optional"/>
            <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"/>
         </extension>
      </complexContent>
   </complexType>

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

   <complexType name="Property" mixed="true">
      <complexContent mixed="true">
         <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" use="optional"
               default="false"/>
            <attribute name="mustSupply" type="boolean" use="optional"
               default="false"/>
            <anyAttribute namespace="##any" processContents="lax"/>
         </extension>
         <!-- extension defines the place to hold default value -->
         <!-- an extension point ; attribute-based only -->
      </complexContent>
   </complexType>

   <complexType name="PropertyValue" mixed="true">
      <complexContent mixed="true">
         <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" use="optional"
               default="false"/>
            <attribute name="source" type="string" use="optional"/>
            <attribute name="file" type="anyURI" use="optional"/>
            <anyAttribute namespace="##any" processContents="lax"/>
         </extension>
         <!-- an extension point ; attribute-based only -->
      </complexContent>
   </complexType>

   <!-- Binding -->
   <element name="binding" type="sca:Binding" abstract="true"/>
```

```
<complexType name="Binding" abstract="true">
   <complexContent>
      <extension base="sca:CommonExtensionBase">
         <sequence>
            <element ref="sca:wireFormat" minOccurs="0" maxOccurs="1" />
            <element ref="sca:operationSelector"
               minOccurs="0" maxOccurs="1" />
            <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"/>
      </extension>
   </complexContent>
</complexType>

<!-- Binding Type -->
<element name="bindingType" type="sca:BindingType"/>
<complexType name="BindingType">
   <complexContent>
      <extension base="sca:CommonExtensionBase">
         <sequence>
            <any namespace="##other" processContents="lax" minOccurs="0"
               maxOccurs="unbounded"/>
         </sequence>
         <attribute name="type" type="QName" use="required"/>
         <attribute name="alwaysProvides" type="sca:listOfQNames"
            use="optional"/>
         <attribute name="mayProvide" type="sca:listOfQNames"
            use="optional"/>
      </extension>
   </complexContent>
</complexType>

<!-- WireFormat Type -->
<element name="wireFormat" type="sca:WireFormatType"/>
<complexType name="WireFormatType" abstract="true">
   <sequence>
      <any namespace="##other" processContents="lax" minOccurs="0"
         maxOccurs="unbounded" />
   </sequence>
   <anyAttribute namespace="##other" processContents="lax"/>
</complexType>

<!-- OperationSelector Type -->
<element name="operationSelector" type="sca:OperationSelectorType"/>
<complexType name="OperationSelectorType" abstract="true">
   <sequence>
      <any namespace="##other" processContents="lax" minOccurs="0"
         maxOccurs="unbounded" />
   </sequence>
   <anyAttribute namespace="##other" processContents="lax"/>
</complexType>
<!-- Callback -->
```

```
   <element name="callback" type="sca:Callback"/>
   <complexType name="Callback">
      <complexContent>
         <extension base="sca:CommonExtensionBase">
            <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"/>
         </extension>
      </complexContent>
   </complexType>

   <!-- Component -->
   <complexType name="Component">
      <complexContent>
         <extension base="sca:CommonExtensionBase">
            <sequence>
               <element ref="sca:implementation" minOccurs="0"/>
               <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"/>
            <attribute name="constrainingType" type="QName" use="optional"/>
            <attribute name="requires" type="sca:listOfQNames"
               use="optional"/>
            <attribute name="policySets" type="sca:listOfQNames"
               use="optional"/>
         </extension>
      </complexContent>
   </complexType>

   <!-- Component Service -->
   <complexType name="ComponentService">
      <complexContent>
         <extension base="sca:Contract">
         </extension>
      </complexContent>
   </complexType>

   <!-- Component Reference -->
   <complexType name="ComponentReference">
      <complexContent>
         <extension base="sca:Contract">
            <attribute name="autowire" type="boolean" use="optional"/>
            <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"/>
         </extension>
      </complexContent>
   </complexType>

   <!-- Component Type Reference -->
   <complexType name="ComponentTypeReference">
      <complexContent>
         <restriction base="sca:ComponentReference">
            <sequence>
               <element ref="sca:documentation" minOccurs="0"
                  maxOccurs="unbounded"/>
               <element ref="sca:interface" minOccurs="0"/>
               <element name="operation" type="sca:Operation" minOccurs="0"
                  maxOccurs="unbounded"/>
               <element ref="sca:binding" minOccurs="0"
                  maxOccurs="unbounded"/>
               <element ref="sca:callback" minOccurs="0"/>
               <any namespace="##other" processContents="lax" minOccurs="0"
                  maxOccurs="unbounded"/>
            </sequence>
            <attribute name="name" type="NCName" use="required"/>
            <attribute name="autowire" type="boolean" use="optional"/>
            <attribute name="wiredByImpl" type="boolean" use="optional"
               default="false"/>
            <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="##other" processContents="lax"/>
         </restriction>
      </complexContent>
   </complexType>

   <!-- Implementation -->
   <element name="implementation" type="sca:Implementation" abstract="true"/>
   <complexType name="Implementation" abstract="true">
      <complexContent>
         <extension base="sca:CommonExtensionBase">
            <attribute name="requires" type="sca:listOfQNames"
               use="optional"/>
            <attribute name="policySets" type="sca:listOfQNames"
               use="optional"/>
         </extension>
      </complexContent>
   </complexType>

   <!-- Implementation Type -->
   <element name="implementationType" type="sca:ImplementationType"/>
   <complexType name="ImplementationType">
      <complexContent>
         <extension base="sca:CommonExtensionBase">
            <sequence>
               <any namespace="##other" processContents="lax" minOccurs="0"
                  maxOccurs="unbounded"/>
            </sequence>
            <attribute name="type" type="QName" use="required"/>
            <attribute name="alwaysProvides" type="sca:listOfQNames"
                  use="optional"/>
            <attribute name="mayProvide" type="sca:listOfQNames"
                  use="optional"/>
         </extension>
      </complexContent>
   </complexType>

   <!-- Wire -->
   <complexType name="Wire">
      <complexContent>
         <extension base="sca:CommonExtensionBase">
            <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"/>
         </extension>
      </complexContent>
   </complexType>

   <!-- Include -->
   <element name="include" type="sca:Include"/>
   <complexType name="Include">
      <complexContent>
         <extension base="sca:CommonExtensionBase">
            <attribute name="name" type="QName"/>
         </extension>
      </complexContent>
   </complexType>

   <!-- Operation -->
   <complexType name="Operation">
      <complexContent>
         <extension base="sca:CommonExtensionBase">
            <attribute name="name" type="NCName" use="required"/>
            <attribute name="requires" type="sca:listOfQNames"
               use="optional"/>
            <attribute name="policySets" type="sca:listOfQNames"
               use="optional"/>
         </extension>
      </complexContent>
   </complexType>

   <!-- Constraining Type -->
   <element name="constrainingType" type="sca:ConstrainingType"/>
   <complexType name="ConstrainingType">
      <complexContent>
         <extension base="sca:CommonExtensionBase">
            <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"/>
         </extension>
      </complexContent>
   </complexType>

   <!-- Intents within WSDL documents -->
   <attribute name="requires" type="sca:listOfQNames"/>

   <!-- Marker for operations ending a conversation -->
   <attribute name="endsConversation" type="boolean" default="false"/>

   <!-- Global attribute definition for @callback to mark a WSDL port type
        as having a callback interface defined in terms of a second port
        type. -->
   <attribute name="callback" type="anyURI"/>

   <!-- Miscellaneous simple type definitions -->
   <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>

   <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"?>
<!-- Copyright(C) OASIS(R) 2005,2008. All Rights Reserved. OASIS trademark,
IPR and other policies apply.  -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
```

```
   targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
   xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
   elementFormDefault="qualified">

   <include schemaLocation="sca-core-1.1-schema-200803.xsd"/>

   <!-- SCA Binding -->
   <element name="binding.sca" type="sca:SCABinding"
      substitutionGroup="sca:binding"/>
   <complexType name="SCABinding">
      <complexContent>
         <extension base="sca:Binding"/>
      </complexContent>
   </complexType>

</schema>
```

A.4 sca-interface-java.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright(C) OASIS(R) 2005,2008. All Rights Reserved. OASIS trademark,
IPR and other policies apply.  -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
   xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
   elementFormDefault="qualified">

   <include schemaLocation="sca-core-1.1-schema-200803.xsd"/>

   <!-- Java Interface -->
   <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"?>
<!-- Copyright(C) OASIS(R) 2005,2008. All Rights Reserved. OASIS trademark,
IPR and other policies apply.  -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
   xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
   elementFormDefault="qualified">

   <include schemaLocation="sca-core-1.1-schema-200803.xsd"/>

   <!-- WSDL Interface -->
   <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"?>
<!-- Copyright(C) OASIS(R) 2005,2008. All Rights Reserved. OASIS trademark,
IPR and other policies apply.  -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
   targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
   elementFormDefault="qualified">

   <include schemaLocation="sca-core-1.1-schema-200803.xsd"/>

   <!-- Java Implementation -->
   <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"/>
            <anyAttribute namespace="##any" processContents="lax"/>
```

```
        </extension>
      </complexContent>
   </complexType>

</schema>
```

A.7 sca-implementation-composite.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright(C) OASIS(R) 2005,2008. All Rights Reserved. OASIS trademark,
IPR and other policies apply.  -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
   targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
   elementFormDefault="qualified">

   <include schemaLocation="sca-core-1.1-schema-200803.xsd"/>

   <!-- Composite Implementation -->
   <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"/>
         </extension>
      </complexContent>
   </complexType>

</schema>
```

A.8 sca-definitions.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright(C) OASIS(R) 2005,2008. All Rights Reserved. OASIS trademark,
IPR and other policies apply.  -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
   xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
   elementFormDefault="qualified">

   <include schemaLocation="sca-core-1.1-schema-200803.xsd"/>
   <include schemaLocation="sca-policy-1.1-schema-200803.xsd"/>

   <!-- Definitions -->
   <element name="definitions" type="sca:tDefinitions"/>
   <complexType name="tDefinitions">
      <complexContent>
         <extension base="sca:CommonExtensionBase">
            <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>
         </extension>
      </complexContent>
   </complexType>

</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]

A.12 sca-contribution.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright(C) OASIS(R) 2005,2008. All Rights Reserved. OASIS trademark,
IPR and other policies apply.  -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
   targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
   elementFormDefault="qualified">

   <include schemaLocation="sca-core-1.1-schema-200803.xsd"/>

   <!-- Contribution -->
   <element name="contribution" type="sca:ContributionType"/>
   <complexType name="ContributionType">
      <complexContent>
         <extension base="sca:CommonExtensionBase">
            <sequence>
               <element name="deployable" type="sca:DeployableType"
                  maxOccurs="unbounded"/>
               <element name="import" type="sca:ImportType" minOccurs="0"
                  maxOccurs="unbounded"/>
               <element name="export" type="sca:ExportType" minOccurs="0"
                  maxOccurs="unbounded"/>
               <any namespace="##other" processContents="lax" minOccurs="0"
                  maxOccurs="unbounded"/>
            </sequence>
         </extension>
      </complexContent>
   </complexType>

   <!-- Deployable -->
   <complexType name="DeployableType">
      <complexContent>
         <extension base="sca:CommonExtensionBase">
            <sequence>
               <any namespace="##other" processContents="lax" minOccurs="0"
                  maxOccurs="unbounded"/>
            </sequence>
            <attribute name="composite" type="QName" use="required"/>
         </extension>
      </complexContent>
   </complexType>

   <!-- Import -->
   <element name="importBase" type="sca:Import" abstract="true" />
   <complexType name="Import" abstract="true">
      <complexContent>
         <extension base="sca:CommonExtensionBase">
            <sequence>
               <any namespace="##other" processContents="lax" minOccurs="0"
                  maxOccurs="unbounded"/>
            </sequence>
         </extension>
      </complexContent>
   </complexType>

   <element name="import" type="sca:ImportType"/>
   <complexType name="ImportType">
      <complexContent>
         <extension base="sca:Import">
            <attribute name="namespace" type="string" use="required"/>
            <attribute name="location" type="anyURI" use="optional"/>
         </extension>
      </complexContent>
   </complexType>

   <!-- Export -->
   <element name="exportBase" type="sca:Export" abstract="true" />
   <complexType name="Export" abstract="true">
      <complexContent>
         <extension base="sca:CommonExtensionBase">
            <sequence>
               <any namespace="##other" processContents="lax" minOccurs="0"
                  maxOccurs="unbounded"/>
            </sequence>
         </extension>
      </complexContent>
   </complexType>

   <element name="export" type="sca:ExportType"/>
   <complexType name="ExportType">
      <complexContent>
         <extension base="sca:Export">
            <attribute name="namespace" type="string" use="required"/>
         </extension>
      </complexContent>
   </complexType>

</schema>
```

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.
How a Service is identified as remotable is dependant on the Component implementation technology used. See the relevant SCA Implementation Specification for more information. As an example, to define a Remotable Service, a Component implemented in Java would have a Java Interface 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.

How a Service is identified as local is dependant on the Component implementation technology used. See the relevant SCA Implementation Specification for more information. As an example, to define a Local Service, a Component implemented in Java would define a Java Interface that does not have 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 a number of interface type systems, for example:

- Java interfaces
- WSDL portTypes
- C, C++ header files

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***.

Valid wire sources are component references. Valid wire targets are component services.

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. Conformance Items

This section contains a list of conformance items for the SCA Assembly specification.

Conformance ID	Description
[ASM13001]	An SCA runtime MUST reject a composite file that does not conform to the sca-core.xsd schema.
[ASM40001]	The extension of a componentType side file name MUST be .componentType.
[ASM40002]	If present, the @constrainingType attribute of a <componentType/> element MUST reference a <constrainingType/> element in the Domain through its QName.
[ASM40003]	The @name attribute of a <service/> child element of a <componentType/> MUST be unique amongst the service elements of that <componentType/>.
[ASM40004]	The @name attribute of a <reference/> child element of a <componentType/> MUST be unique amongst the reference elements of that <componentType/>.
[ASM40005]	The @name attribute of a <property/> child element of a <componentType/> MUST be unique amongst the property elements of that <componentType/>.
[ASM40006]	If @wiredByImpl is set to "true", then any reference targets configured for this reference MUST be ignored by the runtime.
[ASM40007]	The value of the property @type attribute MUST be the QName of an XML schema type.
[ASM40008]	The value of the property @element attribute MUST be the QName of an XSD global element.
[ASM40009]	The SCA runtime MUST ensure that any implementation default property value is replaced by a value for that property explicitly set by a component using that implementation.
[ASM50001]	The @name attribute of a <component/> child element of a <composite/> MUST be unique amongst the component elements of that <composite/>
[ASM50002]	The @name attribute of a service element of a <component/> MUST be unique amongst the service elements of that <component/>
[ASM50003]	The @name attribute of a service element of a <component/> MUST match the @name attribute of a service element of the componentType of the <implementation/> child element of the component.
[ASM50004]	If a <service/> element has an interface subelement specified, the interface MUST provide a compatible subset of the interface declared on the componentType of the implementation
[ASM50005]	If no binding elements are specified for the service, then the bindings specified for the equivalent service in the componentType of the implementation MUST be used, but if the componentType also has no bindings specified, then <binding.sca/> MUST be used as the binding. If binding elements are specified for the service, then those bindings MUST be used and they override any bindings specified for the equivalent service in the componentType of the implementation.
[ASM50006]	If the callback element is present and contains one or more binding child elements, then those bindings MUST be used for the callback.
[ASM50007]	The @name attribute of a service element of a <component/> MUST be unique amongst the service elements of that <component/>
[ASM50008]	The @name attribute of a reference element of a <component/> MUST match the @name attribute of a reference element of the componentType of the

	<implementation/> child element of the component.
[ASM50009]	The value of multiplicity for a component reference MUST only be equal or further restrict any value for the multiplicity of the reference with the same name in the componentType of the implementation, where further restriction means 0..n to 0..1 or 1..n to 1..1.
[ASM50010]	If @wiredByImpl="true" is set for a reference, then the reference MUST NOT be wired statically within a composite, but left unwired.
[ASM50011]	If an interface is declared for a component reference it MUST provide a compatible superset of the interface declared for the equivalent reference in the componentType of the implementation, i.e. provide the same operations or a superset of the operations defined by the implementation for the reference.
[ASM50012]	If no binding elements are specified for the reference, then the bindings specified for the equivalent reference in the componentType of the implementation MUST be used, but if the componentType also has no bindings specified, then <binding.sca/> MUST be used as the binding. If binding elements are specified for the reference, then those bindings MUST be used and they override any bindings specified for the equivalent reference in the componentType of the implementation.
[ASM50013]	If @wiredByImpl="true", other methods of specifying the target service MUST NOT be used.
[ASM50014]	If @autowire="true", the autowire procedure MUST only be used if no target is identified by any of the other ways listed above. It is not an error if @autowire="true" and a target is also defined through some other means, however in this case the autowire procedure MUST NOT be used.
[ASM50015]	If a binding element has a value specified for a target service using its @uri attribute, the binding element MUST NOT identify target services using binding specific attributes or elements.
[ASM50016]	It is possible that a particular binding type MAY require that the address of a target service uses more than a simple URI. In such cases, the @uri attribute MUST NOT be used to identify the target service - instead, binding specific attributes and/or child elements must be used.
[ASM50018]	A reference with multiplicity 0..1 or 0..n MAY have no target service defined.
[ASM50019]	A reference with multiplicity 0..1 or 1..1 MUST NOT have more that one target service defined.
[ASM50020]	A reference with multiplicity 1..1 or 1..n MUST have at least one target service defined.
[ASM50021]	A reference with multiplicity 0..n or 1..n MAY have one or more target services defined.
[ASM50022]	Where it is detected that the rules for the number of target services for a reference have been violated, either at deployment or at execution time, an SCA Runtime MUST generate an error no later than when the reference is invoked by the component implementation.
[ASM50023]	Some reference multiplicity errors can be detected at deployment time. In these cases, an error SHOULD be generated by the SCA runtime at deployment time.
[ASM50024]	Other reference multiplicity errors can only be checked at runtime. In these cases, the SCA runtime MUST generate an error no later than when the reference is invoked by the component implementation.
[ASM50025]	Where a component reference is promoted by a composite reference, the promotion MUST be treated from a multiplicity perspective as providing 0 or more target services for the component reference, depending upon the further configuration of the composite reference. These target services are in addition to any target services identified on the component reference itself, subject to the rules relating to multiplicity.
[ASM50026]	If a reference has a value specified for one or more target services in its @target attribute, there MUST NOT be any child <binding/> elements declared for that reference.

[ASM50027] If the @value attribute of a component property element is declared, the type of the property MUST be an XML Schema simple type and the @value attribute MUST contain a single value of that type.

[ASM50028] If the value subelement of a component property is specified, the type of the property MUST be an XML Schema simple type or an XML schema complex type.

[ASM50029] If a component property value is declared using a child element of the <property/> element, the type of the property MUST be an XML Schema global element and the declared child element MUST be an instance of that global element.

[ASM50030] A <component/> element MUST NOT contain two <property/> subelements with the same value of the @name attribute.

[ASM50031] The name attribute of a component property MUST match the name of a property element in the component type of the component implementation.

[ASM50032 If a property is single-valued, the <value/> subelement MUST NOT occur more than once.

[ASM50033] A property <value/> subelement MUST NOT be used when the @value attribute is used to specify the value for that property.

[ASM50034] If any <wire/> element with its @replace attribute set to "true" has a particular reference specified in its @source attribute, the value of the @target attribute for that reference MUST be ignored and MUST NOT be used to define target services for that reference.

[ASM60001] A composite name must be unique within the namespace of the composite.

[ASM60002] @local="true" for a composite means that all the components within the composite MUST run in the same operating system process.

[ASM60003] The name of a composite <service/> element MUST be unique across all the composite services in the composite.

[ASM60004] A composite <service/> element's promote attribute MUST identify one of the component services within that composite.

[ASM60005] If a composite service ***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.

[ASM60006] The name of a composite <reference/> element MUST be unique across all the composite references in the composite.

[ASM60007] Each of the URIs declared by a composite reference's @promote attribute MUST identify a component reference within the composite.

[ASM60008] the interfaces of the component references promoted by a composite reference MUST be the same, or if the composite reference itself declares an interface then all the component reference interfaces must be compatible with the composite reference interface. Compatible means that the component reference interface is the same or is a strict subset of the composite reference interface.

[ASM60009] the intents declared on a composite reference and on the component references which it promoites MUST NOT be mutually exclusive.

[ASM60010] If any intents in the set which apply to a composite reference are mutually exclusive then the SCA runtime MUST raise an error.

[ASM60011] The value specified for the ***multiplicity*** attribute of a composite reference MUST be compatible with the multiplicity specified on each of the promoted component references, i.e. the multiplicity has to be equal or further restrict. So multiplicity 0..1 can be used where the promoted component reference has multiplicity 0..n, multiplicity 1..1 can be used where the promoted component reference has multiplicity 0..n or 1..n and multiplicity 1..n can be used where the promoted component reference has multiplicity 0..n., 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.

[ASM60012] If a composite reference has an ***interface*** specified, it MUST provide an interface which is the same or which is a compatible superset of the interface(s) declared

	by the promoted component reference(s), i.e. provide a superset of the operations in the interface defined by the component for the reference.
[ASM60013]	If no interface is declared on a composite reference, the interface from one of its promoted component references is used, which MUST be the same as or a compatible superset of the interface(s) declared by the promoted component reference(s).
[ASM60014]	The name attribute of a composite property MUST be unique amongst the properties of the same composite.
[ASM60015]	the source interface and the target interface of a wire MUST either both be remotable or else both be local
[ASM60016]	the operations on the target interface of a wire MUST be the same as or be a superset of the operations in the interface specified on the source
[ASM60017]	compatibility between the source interface and the target interface for a wire for the individual operations is defined as compatibility of the signature, that is operation name, input types, and output types MUST be the same.
[ASM60018]	the order of the input and output types for operations in the source interface and the target interface of a wire also MUST be the same.
[ASM60019]	the set of Faults and Exceptions expected by each operation in the source interface MUST be the same or be a superset of those specified by the target interface.
[ASM60020]	other specified attributes of the source interface and the target interface of a wire MUST match, including Scope and Callback interface
[ASM60021]	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.
[ASM60022]	For each component reference for which autowire is enabled, the the SCA runtime MUST search within the composite for target services which are compatible with the reference.
[ASM60023]	the target service interface MUST be a compatible superset of the reference interface when using autowire to wire a reference (as defined in the section on Wires)
[ASM60024]	the intents, and policies applied to the service MUST be compatible with those on the reference when using autowire to wire a reference – so that wiring the reference to the service will not cause an error due to policy mismatch
[ASM60025]	for an autowire reference with multiplicity 0..1 or 1..1, the SCA runtime MUST wire the reference to one of the set of valid target services chosen from the set in a runtime-dependent fashion
[ASM60026]	for an autowire reference with multiplicity 0..n or 1..n, the reference MUST be wired to all of the set of valid target services
[ASM60027]	for an autowire reference with multiplicity 0..1 or 0..n, if the SCA runtime finds no valid target service, there is no problem – no services are wired and the SCA runtime MUST NOT raise an error
[ASM60028]	for an autowire reference with multiplicity 1..1 or 1..n, if the SCA runtime finds no valid target services an error MUST be raised by the SCA runtime since the reference is intended to be wired
[ASM60030]	The @name attribute of an <implementation.composite/> element MUST contain the QName of a composite in the SCA Domain.
[ASM60031]	The SCA runtime MUST raise an error if the composite resulting from the inclusion of one composite into another is invalid.
[ASM60032]	For a composite used as a component implementation, each composite service offered by the composite MUST promote a component service of a component that is within the composite.
[ASM60033]	For a composite used as a component implementation, every component reference of components within the composite with a multiplicity of 1..1 or 1..n MUST be wired or promoted

(according to the various rules for specifying target services for a component reference described in section 5.3.1).

[ASM60034] For a composite used as a component implementation, all properties of components within the composite, where the underlying component implementation specifies "mustSupply=true" for the property, MUST either specify a value for the property or source the value from a composite property.

[ASM70001] The constrainingType specifies the services, references and properties that MUST be implemented by the implementation of the component to which the constrainingType is attached.

[ASM70002] If the configuration of the component or its implementation do not conform to the constrainingType specified on the component element, the SCA runtime MUST raise an error.

[ASM70003] The name attribute of the constraining type MUST be unique in the SCA domain.

[ASM70004] When an implementation is constrained by a constrainingType its component type MUST contain all the services, references and properties specified in the constrainingType.

[ASM70005] An implementation MAY contain additional services, additional optional references (multiplicity 0..1 or 0..n) and additional optional properties beyond those declared in the constraining type, but MUST NOT contain additional non-optional references (multiplicity 1..1 or 1..n) or additional non-optional properties (a property with mustSupply=true).

[ASM70006] Additional services, references and properties provided by the implementation which are not declared in the constrainingType associated with a component MUST NOT be configured in any way by the containing composite.

[ASM70007] A component or implementation can use a qualified form of an intent specified in unqualified form in the constrainingType, but if the constrainingType uses the qualified form of an intent, then the component or implementation MUST also use the qualified form, otherwise there is an error.

[ASM80001] The interface.wsdl @interface attribute MUST reference a portType of a WSDL 1.1 document.

[ASM80002] Remotable service Interfaces MUST NOT make use of ***method or operation overloading***.

[ASM80003] If a remotable service is called locally or remotely, the SCA container MUST ensure sure that no modification of input messages by the service or post-invocation modifications to return messages are seen by the caller.

[ASM80004] 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 MUST provide an implementation of the callback interface.

[ASM80005] 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.

[ASM80006] Where a service or a reference has a conversational interface, the conversational intent MUST be attached either to the interface itself, or to the service or reference using the interface.

[ASM80007] Once an operation marked with endsConversation has been invoked, any subsequent attempts to call an operation or a callback operation associated with the same conversation MUST generate a sca:ConversationViolation fault.

[ASM80008] Any service or reference that uses an interface marked with required intents MUST implicitly add those intents to its own @requires list.

[ASM80009] In a bidirectional interface, the service interface can have more than one operation defined, and the callback interface can also have more than one operation defined. SCA runtimes MUST allow an invocation of any operation on the service interface to be followed by zero, one or many invocations of any of the operations on the callback interface.

[ASM80010] Whenever an interface document declaring a callback interface is used in the

	declaration of an <interface/> element in SCA, it MUST be treated as being bidirectional with the declared callback interface.
[ASM80011]	If an <interface/> element references an interface document which declares a callback interface and also itself contains a declaration of a callback interface, the two callback interfaces MUST be compatible.
[ASM80012]	Where a component uses an implementation and the component configuration explicitly declares an interface for a service or a reference, if the matching service or reference declaration in the component type declares an interface which has a callback interface, then the component interface declaration MUST also declare a compatible interface with a compatible callback interface.
[ASM80013]	If the service or reference declaration in the component type declares an interface without a callback interface, then the component configuration for the corresponding service or reference MUST NOT declare an interface with a callback interface.
[ASM80014]	Where a composite declares an interface for a composite service or a composite reference, if the promoted service or promoted reference has an interface which has a callback interface, then the interface declaration for the composite service or the composite reference MUST also declare a compatible interface with a compatible callback interface.
[ASM80015]	If the promoted service or promoted reference has an interface without a callback interface, then the interface declaration for the composite service or composite reference MUST NOT declare a callback interface.
[ASM80016]	The interface.wsdl @callbackInterface attribute, if present, MUST reference a portType of a WSDL 1.1 document.
[ASM90001]	For a binding of a ***reference*** the URI attribute defines the target URI of the reference. This MUST be either the componentName/serviceName for a wire to an endpoint within the SCA domain, or the accessible address of some service endpoint either inside or outside the SCA domain (where the addressing scheme is defined by the type of the binding).
[ASM90002]	When a service or reference has multiple bindings, only one binding can have the default name value; all others must have a name value specified that is unique within the service or reference.
[ASM90003]	If a reference has any bindings they MUST be resolved which means that each binding MUST include a value for the @URI attribute or MUST otherwise specify an endpoint. The reference MUST NOT be wired using other SCA mechanisms.
[ASM90004]	a wire target MAY be specified with a syntax of "componentName/serviceName/bindingName".
[ASM10001]	all of the QNames for the definitions contained in definitions.xml files MUST be unique within the domain.
[ASM12001]	For any contribution 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
[ASM12002]	Within any contribution packaging A directory resource SHOULD exist at the root of the hierarchy named META-INF
[ASM12003]	Within any contribution packaging 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.
[ASM12004]	Optionally, in the sca-contribution.xml file, additional elements MAY exist that list the namespaces of constructs that are needed by the contribution and which are be found elsewhere, for example in other contributions.
[ASM12005]	Where present, these mechanisms MUST be used by the SCA runtime to resolve artifact dependencies.
[ASM12006]	SCA requires that all runtimes MUST support the ZIP packaging format for contributions.
[ASM12007]	Implementations of SCA MAY also generate an error if there are conflicting names exported from multiple contributions.

[ASM12008] SCA runtimes MAY choose not to provide the contribution functions functionality in any way.

[ASM12009] 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.

[ASM12010] Where present, non-SCA artifact resolution mechanisms MUST be used by the SCA runtime in precendence to the SCA mechanisms.

[ASM12011] If one of the non-SCA artifact resolution 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.

[ASM12012] The value of @autowire for the logical domain composite MUST be autowire="false".

[ASM12013] For components at the Domain level, with References for which @autowire="true" applies, the behaviour of the SCA runtime for a given Domain MUST take ONE of the 3 following forms:

1) The SCA runtime MAY disallow deployment of any components with autowire References. In this case, the SCA runtime MUST generate an exception at the point where the component is deployed.

2) The SCA runtime MAY evaluate the target(s) for the reference at the time that the component is deployed and not update those targets when later deployment actions occur.

3) The SCA runtime MAY re-evaluate the target(s) for the reference dynamically as later deployment actions occur resulting in updated reference targets which match the new Domain configuration. How the new configuration of the reference takes place is described by the relevant client and implementation specifications.

[ASM12014] Where <wire/> elements are added, removed or replaced by deployment actions, the components whose references are affected by those deployment actions MAY have their references updated by the SCA runtime dynamically without the need to stop and start those components.

[ASM12015] Where components are updated by deployment actions (their configuration is changed in some way, which may include changing the wires of component references), the new configuration MUST apply to all new instances of those components once the update is complete.

[ASM12016] An SCA runtime MAY choose to maintain existing instances with the old configuration of components updated by deployment actions, but an SCA runtime MAY choose to stop and discard existing instances of those components.

[ASM12017] Where a component that is the target of a wire is removed, without the wire being changed, then future invocations of the reference that use that wire SHOULD fail with a ServiceUnavailable fault. If the wire is the result of the autowire process, the SCA runtime MUST:

- either cause future invocation of the target component's services to fail with a ServiceUnavailable fault
- or alternatively, if an alternative target component is available that satisfies the autowire process, update the reference of the source component

[ASM12018] Where a component that is the target of a wire is updated, future invocations of that reference SHOULD use the updated component.

[ASM12019] Where an existing domain level component is updated, an SCA runtime MAY maintain a copy of a component offering a conversational service until all existing conversations complete - alternatively all existing conversations MAY be terminated.

[ASM12020] Where a component is added to the domain that is a potential target for a domain level component reference where that reference is marked as @autowire=true, the SCA runtime MUST:

- either update the references for the source component once the new component is running.

or alternatively, defer the updating of the references of the source component until

	the source component is stopped and restarted.
[ASM12021]	The SCA runtime MUST raise an error if an artifact cannot be resolved using these mechanisms, if present.
[ASM12022]	There can be multiple import declarations for a given namespace. Where multiple import declarations are made for the same namespace, all the locations specified MUST be searched in lexical order.
[ASM12023]	When a contribution contains a reference to an artifact from a namespace that is declared in an import statement of the contribution, if the SCA artifact resolution mechanism is used to resolve the artifact, the SCA runtime MUST resolve artifacts in the following order: 1. from the locations identified by the import statement(s) for the namespace. Locations MUST NOT be searched recursively in order to locate artifacts (ie only a one-level search is performed). 2. from the contents of the contribution itself.
[ASM12024]	The SCA runtime MUST ignore local definitions of an artifact if the artifact is found through resolving an import statement.
[ASM12025]	The SCA runtime MUST raise an error if an artifact cannot be resolved by the precedence order above.

D. 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]

E. Non-Normative Text

F. 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
2	2008-01-04	Michael Beisiegel	composite section - changed order of subsections from property, reference, service to service, reference, property - progressive disclosure of pseudo schemas, each section only shows what is described - attributes description now starts with name : type (cardinality) - child element description as list, each item starting with name : type (cardinality) - added section in appendix to contain complete pseudo schema of composite - moved component section after implementation section - made the ConstrainingType section a top level section - moved interface section to after constraining type section component section - added subheadings for Implementation, Service, Reference, Property - progressive disclosure of pseudo schemas, each section only shows what is described - attributes description now starts with name : type (cardinality) - child element description as list, each item starting with name : type (cardinality) implementation section - changed title to “Implementation and ComponentType” - moved implementation instance related stuff from implementation section to component implementation section - added subheadings for Service, Reference, Property, Implementation - progressive disclosure of pseudo schemas, each section only shows what is described - attributes description now starts with name : type (cardinality) - child element description as list, each item starting with name : type (cardinality) - attribute and element description still needs to be completed, all implementation statements

			on services, references, and properties should go here - added complete pseudo schema of componentType in appendix - added “Quick Tour by Sample” section, no content yet - added comment to introduction section that the following text needs to be added "This specification is efined in terms of infoset and not XML 1.0, even though the spec uses XML 1.0/1.1 terminology. A mapping from XML to infoset (... link to infoset specification ...) is trivial and should be used for non-XML serializations."
3	2008-02-15	Anish Karmarkar Michael Beisiegel	Incorporated resolutions from 2008 Jan f2f. - issue 9 - issue 19 - issue 21 - issue 4 - issue 1A - issue 27 - in Implementation and ComponentType section added attribute and element description for service, reference, and property - removed comments that helped understand the initial restructuring for WD02 - added changes for issue 43 - added changes for issue 45, except the changes for policySet and requires attribute on property elements - used the NS http://docs.oasis-open.org/ns/opencsa/sca/200712 - updated copyright stmt - added wordings to make PDF normative and xml schema at the NS uri autoritative
4	2008-04-22	Mike Edwards	Editorial tweaks for CD01 publication: - updated URL for spec documents - removed comments from published CD01 version - removed blank pages from body of spec
5	2008-06-30	Anish Karmarkar Michael Beisiegel	Incorporated resolutions of issues: 3, 6, 14 (only as it applies to the component property element), 23, 25, 28, 25, 38, 39, 40, 42, 45 (except for adding @requires and @policySets to property elements), 57, 67, 68, 69
6	2008-09-23	Mike Edwards	Editorial fixes in response to Mark Combellack's review contained in email: http://lists.oasis-open.org/archives/sca-assembly/200804/msg00089.html
7 CD01 - Rev3	2008-11-18	Mike Edwards	• Specification marked for conformance statements. New Appendix (D) added

			containing a table of all conformance statements. Mass of related minor editorial changes to remove the use of RFC2119 words where not appropriate.
8 CD01 - Rev4	2008-12-11	Mike Edwards	- Fix problems of misplaced statements in Appendix D - Fixed problems in the application of Issue 57 - section 5.3.1 & Appendix D as defined in email: http://lists.oasis-open.org/archives/sca-assembly/200811/msg00045.html - Added Conventions section, 1.3, as required by resolution of Issue 96. - Issue 32 applied - section B2 - Editorial addition to section 8.1 relating to no operation overloading for remotable interfaces, as agreed at TC meeting of 16/09/2008.
9 CD01 - Rev5	2008-12-22	Mike Edwards	- Schemas in Appendix B updated with resolutions of Issues 32 and 60 - Schema for contributions - Appendix B12 - updated with resolutions of Issues 53 and 74. - Issues 53 and 74 incorporated - Sections 11.4, 11.5
10 CD01-Rev6	2008-12-23	Mike Edwards	- Issues 5, 71, 92 - Issue 14 - remaining updates applied to ComponentType (section 4.1.3) and to Composite Property (section 6.3)
11 CD01-Rev7	2008-12-23	Mike Edwards	All changes accepted before revision from Rev6 started - due to changes being applied to previously changed sections in the Schemas Issues 12 & 18 - Section B2 Issue 63 - Section C3 Issue 75 - Section C12 Issue 65 - Section 7.0 Issue 77 - Section 8 + Appendix D Issue 69 - Sections 5.1, 8 Issue 45 - Sections 4.1.3, 5.4, 6.3, B2. Issue 56 - Section 8.2, Appendix D Issue 41 - Sections 5.3.1, 6.4, 12.7, 12.8, Appendix D
12 CD01-Rev8	2008-12-30	Mike Edwards	Issue 72 - Removed Appendix A Issue 79 - Sections 9.0, 9.2, 9.3, Appendix A.2 Issue 62 - Sections 4.1.3, 5.4 Issue 26 - Section 6.5 Issue 51 - Section 6.5 Issue 36 - Section 4.1 Issue 44 - Section 10, Appendix C Issue 89 - Section 8.2, 8.5, Appendix A, Appendix C Issue 16 - Section 6.8, 9.4 Issue 8 - Section 11.2.1 Issue 17 - Section 6.6 Issue 30 - Sections 4.1.1, 4.1.2, 5.2, 5.3, 6.1, 6.2, 9 Issue 33 - insert new Section 8.4
12 CD01-Rev8a	2009-01-13	Bryan Aupperle Mike Edwards	Issue 99 - Section 8

13 CD02	2009-01-14	Mike Edwards	All changes accepted All comments removed