Service Component Architecture Assembly Model Specification Version 1.2

Committee Specification Draft 01

19 July 2011

Specification URIs

This version:

http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-spec/v1.2/csd01/sca-assembly-spec-v1.2-csd01.pdf (Authoritative)
http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-spec/v1.2/csd01/sca-assembly-spec-v1.2-csd01.html
http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-spec/v1.2/csd01/sca-assembly-spec-v1.2-csd01.doc

Previous version:

N/A

Latest version:

http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-spec/v1.2/sca-assembly-spec-v1.2.pdf (Authoritative)
http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-spec/v1.2/sca-assembly-spec-v1.2.html
http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-spec/v1.2/sca-assembly-spec-v1.2.doc

Technical Committee:

OASIS Service Component Architecture / Assembly (SCA-Assembly) TC

Chairs:

Martin Chapman (martin.chapman@oracle.com), Oracle
Mike Edwards (mike_edwards@uk.ibm.com), IBM

Editors:

Michael Beisiegel (mbgl@us.ibm.com), IBM
Anish Karmarkar (Anish.Karmarkar@oracle.com), Oracle
Sanjay Patil (sanjay.patil@sap.com), SAP
Michael Rowley (michael.rowley@activevos.com), Active Endpoints

Additional artifacts:

This prose specification is one component of a Work Product which also includes:

- XML schemas: http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-spec/v1.2/csd01/xsd/

Related work:

This specification replaces or supersedes:

- *Service Component Architecture Assembly Model Specification Version 1.1*. Latest version. http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-spec-v1.1.html
- *Service Component Architecture Assembly Model Specification Version 1.00*. March 15, 2007. http://www.osoa.org/download/attachments/35/SCA_AssemblyModel_V100.pdf

This specification is related to:

- *SCA Policy Framework Version 1.1*. Latest version. http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1.html

Declared XML namespaces:

- http://docs.oasis-open.org/ns/opencsa/sca/200912

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. In addition, SCA also provides a model for organizing components that produce and consume events and the processing of such events.

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
- A model for event processing and Pub/Sub -- a particular style of organizing components that produce and consume events in which the producing components are decoupled from the consuming components

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” 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).

Citation format:

When referencing this specification the following citation format should be used:

[SCA-Assembly-v1.2]

Service Component Architecture Assembly Model Specification Version 1.2. 19 July 2011. OASIS Committee Specification Draft 01. http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-spec/v1.2/csd01/sca-assembly-spec-v1.2-csd01.html.

Notices

This document and the information contained herein is provided on an "AS IS" basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

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 name "OASIS" is a trademark 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
- A model for event processing and pub/sub -- a particular style of organizing components that produce and consume events in which the producing components are decoupled from the consuming components

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 it is suggested that this is 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*,

IETF RFC 2119, March 1997.

http://www.ietf.org/rfc/rfc2119.txt

[SCA-Java]

OASIS Committee Draft 03, "SCA *POJO Component Implementation Specification Version 1.1*", November 2010

http://docs.oasis-open.org/opencsa/sca-j/sca-javaci-1.1-spec-csprd03.pdf

[SCA-Common-Java]

OASIS Committee Draft 05, "SCA *Java Common Annotations and APIs Specification Version 1.1*", November 2010

http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-csd05.pdf

[SCA BPEL]

OASIS Committee Draft 02, "*SCA WS-BPEL Client and Implementation Specification Version 1.1*", March 2009

http://docs.oasis-open.org/opencsa/sca-bpel/sca-bpel-1.1-spec-cd02.pdf2.pdf

[WSDL-11]

WSDL Specification version 1.1
http://www.w3.org/TR/wsdl

[SCA-WSBINDING]

OASIS Committee Draft 04, "*SCA Web Services Binding Specification Version 1.1*", May 2010
http://docs.oasis-open.org/opencsa/sca-bindings/sca-wsbinding-1.1-spec-cd04.pdf

[SCA-POLICY]

OASIS Committee Draft 04, "*SCA Policy Framework Specification Version 1.1*", September 2010
http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1.pdf

[SCA-JMSBINDING]

OASIS Committee Draft 05, "*SCA JMS Binding Specification Version 1.1 Version 1.1*", November 2010
http://docs.oasis-open.org/opencsa/sca-bindings/sca-jmsbinding-1.1-spec-csprd03.pdf

[SCA-CPP-Client]

OASIS Committee Draft 06, "SCA *Client and Implementation for C++ Specification Version 1.1*", October 2010
http://docs.oasis-open.org/opencsa/sca-c-cpp/sca-cppcni-1.1-spec-cd06.pdf

[SCA-C-Client]

OASIS Committee Draft 06, "*SCA Client and Implementation for C Specification Version 1.1*", October 2010
http://docs.oasis-open.org/opencsa/sca-c-cpp/sca-ccni-1.1-spec-cd06.pdf

[ZIP-FORMAT]

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

[XML-INFOSET]

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

[WSDL11_Identifiers]

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

[SCA-TSA]

OASIS Committee Draft 01, "*Test Suite Adaptation for SCA Assembly Model Version 1.1 Specification*", July 2010
http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-testsuite-adaptation-cd01.pdf

[SCA-IMPLTYPDOC]

OASIS Committee Draft 01, "*Implementation Type Documentation Requirements for SCA Assembly Model Version 1.1 Specification*", July 2010

http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-impl-type-documentation-cd01.pdf

[SCA-ASSEMBLY-11]

OASIS Committee Specification Draft 08, "Service Component Architecture *Assembly Model Specification Version 1.1*", June 2011

http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-csd08.pdf

[WS-ED]

W3C Candidate Recommendation, "Web Services Event Descriptions (WS-EventDescriptions)", April 2011

http://www.w3.org/TR/2011/CR-ws-event-descriptions-20110428

1.3 Non-Normative References

[SDO]

OASIS Committee Draft 02, "*Service Data Objects Specification Version 3.0*", November 2009

http://www.oasis-open.org/committees/download.php/35313/sdo-3.0-cd02.zip

[JAX-WS]

JAX-WS Specification

http://jcp.org/en/jsr/detail?id=224

[WSI-BP]

WS-I Basic Profile

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

[WSI-BSP]

WS-I Basic Security Profile

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

[WS-BPEL]

OASIS Standard, "*Web Services Business Process Execution Language Version 2.0*", April 2007

http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

[SCA-ASSEMBLY-TC]

OASIS Committee Draft, " TestCases for the SCA Assembly Model Version 1.1 Specification", August 2010.

http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-testcases.pdf

1.4 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.
  e.g. <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.

1.5 Testcases

The TestCases for the SCA Assembly Model Version 1.1 Specification **[SCA-ASSEMBLY-TC]** defines the TestCases for the SCA Assembly specification.The TestCases represent a series of tests that SCA runtimes are expected to pass in order to claim conformance to the requirements of the SCA Assembly specification.

1.6 Changes in Version 1.2

This version applies all the errata against version 1.1 **[SCA-ASSEMBLY-11]**, and adds support for ***Event Processing*** and ***Pub/Sub*** in the SCA Assembly Model, which deals with:

- ***Event Processing***, which is computing that performs operations on events, including creating, reading, transforming, and deleting events or event objects/representations. Event Processing components interact by creating event messages, which are then distributed to other Event Processing components. An Event Processing component can, in addition, interact with other SCA components using SCA's regular service invocation mechanisms.
- ***Publication*** and ***Subscription*** (often shortened to ***Pub/Sub***), which is a particular style of organizing the components which produce and consume events in which the producing components are decoupled from the consuming components. Components that are interested in consuming events specify their interest through a subscription rather than an interface. The same event may be received by multiple subscribers.

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. In addition, SCA also provides a model for organizing components that produce and consume events and the processing of such events.

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. To support the Pub/Sub style of interaction, an implementation can also have ***consumers*** and ***producers***. Consumers and producers can specify the kind of events that they are interested in consuming or producing, respectively. The component ***configures*** the implementation by providing values for the properties, by wiring the references to services provided by other components, and by connecting the producers/consumers to channels.

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, consumers, producers, channels, and 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, producers and depending on references, consumers, 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 might 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 Event Processing and Pub/Sub Overview

In addition to the ***Service-Reference Model*** described above, the SCA ***Assembly Model*** also supports the ***Pub-Sub Model*** which can be used by components to communicate with each other. With ***service invocation***, one component, the client, invokes an operation on a service reference, which causes that operation to be invoked on a second component, the service provider. The significant charateristics of service invocation are that:

- Each invocation by the client on a reference operation causes one invocation of the operation on one service provider
- The operation itself typically has some implied semantics – the client is expecting some specific task to be performed by the service provider, possibly involving specific data being returned by the provider
- A particular operation is typically grouped with a set of other related operations, as defined by an interface, which as a whole make up the service offered by the provider. The need to implement the interface as a whole is a requirement for the code implementing the components. There is also a requirement that the complete set of operations declared on a reference is supplied by the service provider.
- The provider may respond to the operation invocation with zero or more response messages. These messages may be returned synchronously or asynchronously, but they are returned to the client component that made the original invocation. That they are returned is part of the service contract between the client and the provider

In contrast, in ***event processing*** applications one component, the producer, creates a message called an event, which is sent out and can be received by any number of other components, called consumers. The significant characteristics of this mechanism are that:

- Each event created by a producer may be received by zero, one or many consumer components. The producer is unaware of the specific consumers or the number of consumers that receive any event.
- The consumer cannot respond to an event received – there is in principle no knowledge of the producer component and no route provided by which a response message could be sent to it. The component receiving an event can in turn send out events (or invoke services), but there is no implication that the original producer component will receive any of those events.
- What is done when a consumer receives an event has no implied semantics – the consumer can do what it likes with the event and there are no semantics agreed with the producer
- There is no requirement that a consumer consumes all of the event types that can be produced by a given producer. Neither is there a requirement that a producer produces all of the event types that can be consumed by a consumer. Unlike services, there is no matching of an interface on the producer to an interface on the consumer.
- There is also no direct relationship between event types and the implementation operations or methods used to produce or consume them – e.g., a single operation can handle one event type or many event types, as desired by the writer of the implementation code.
- A consumer can filter which events it is prepared to accept – there is no guarantee that it actually does anything with a given event. The filtering may be on the event type or on the business data within the event or on other metadata associated with the event.

Service operations which are ***one-way*** are close in nature to the sending and receiving of events, but it is notable that for one-way service operations the client component must be aware of the number of target services (multiplicity 0..n or 1..n specified) and the client has to call the operation once for each target. For an event, the producer component simply sends a single event once through its producer – the event is sent to all the consumer components that have expressed interest in that event and are connected (including none), without the producer component being aware of the number or of the recipients.

Event processing involves more loosely-coupled method of combining components into an application than using service interfaces. Events place fewer requirements on the components at each end of the communication. Effectively, in event processing it is only the event types that are shared between the producers and the consumers.

Loose coupling is futher emphasized through the use of ***Pub/Sub***. With Pub/Sub, producers are not connected directly to any consumers – instead, a group of zero or more producers is connected with a group of zero or more consumers through a logical intermediary, called a ***Channel***. The producers publish events to the channel and the consumers receive events from the channel. The actual origin of an event received by a consumer can be any of the producers – without the consumer being directly connected to any of the producers.

In SCA event processing, component implementations may have zero or more ***producers*** and zero or more ***consumers***. The producers and consumers can indicate which event type(s) they deal with. SCA components configure implementations to express where producer events are published to and where consumer events are subscribed from.

2.1.1 Terminology

- *event* – a message sent to zero or more parties that contains information about a situation that has occurred
- *producer* - entity that creates events
- *consumer* - entity that receives events
- *subscription* - records a consumer’s interest in receiving specific kinds of events from some location
- *source* – the place from which a consumer receives events
- *target* – the place to which a producer sends events
- *publication* – the sending of an event from a producer to some targets
- *event type* – every event instance can have an associated event type. Each event type is identified by a unique QName and has an associated shape that can be described using XML Schema global element declaration, and optionally, constraints on the event instance
- *channel* –a mechanism to connect a set of producers with a set of consumers
- *filter* - a mechanism for refining the set of events received by a consumer. A filter may operate on business data within the event itself, or on metadata about the event.

2.1.2 Connections from Producers to Consumers

In SCA, events flow from producers to consumers along logical routes that are defined by the configuration of composites and the components and channels they contain. In particular, components configure producers by declaring targets for the events created by the producer. Components configure consumers by specifying sources for the events received by the consumer and specifying the kind of events that are of interest.

2.1.2.1 Linking Producers to Consumers

Event producers can be linked to event consumers via a third party called a channel, where producers are configured with the channel as a target and consumers are configured with the channel as a source. Using this mechanism, producers and consumers are not directly connected. It is also possible for the producer(s) to connect to a Domain channel (see the Section on Scopes of Channels) at a different time than when the consumer(s) connect to the same channel.

A producer declares where the messages it produces are sent through a list of one or more target URIs in its ***@target*** attribute. The form of the target URIs include:

- The URI of a channel in the same composite as the producer, in the form ***channelName***
- The URI of of a channel at the Domain level in the form ***/channelName***

A consumer declares the sources for the messages it receives through a list of one or more source URIs in its **@*source*** attribute. The form of the source URIs include:

- The URI of a channel in the same composite in the form ***channelName***
- The URI of a channel at the Domain level in the form ***/channelName***

2.1.2.2 Producers, Consumers, and Composites

When an assembler creates a composite that is intended for use as an implementation, the assembler can decide whether consumers and producers within the composite are visible outside the scope of the composite or not.

The assembler can also decide on what level of control is given to the higher level component that is using the composite as its implementation – i.e., the assembler can decide what appears in the component type of the composite, which can then be configured by the higher level component.

One technique which enables component producers to send events outside the composite and for component consumers to receive events from outside the composite is to configure producers and/or consumers of components inside the composite to use domain channels – that is, channels at the Domain level. See the Section Scopes of Channels for more details on domain channels. This approach "hard wires" the producers and consumers within the composite - the higher level component cannot reconfigure the sources and targets.

An alternative technique for configuring a component producer element is to declare a ***composite producer*** element which ***promotes*** the component producer. Similarly a component consumer can be configured by declaring a ***composite consumer*** element which ***promotes*** the component consumer. When producers and consumers are promoted in this way, and the composite is used as the implementation of some higher level component, the assembler of the higher level composite can control where the events flow to and from, through configuration of the higher level component. This technique promotes reuse of the lower level composite in different contexts.

Each producer and consumer can be connected to zero or more channels. If a producer is not connected then any events it produces are discarded and are not received by any consumer. If a consumer is not connected, then it never receives any events.

A Composite can contain one or more Channels. Events can be sent to a channel by producers within the composite, if connected, and events may be received from a channel by consumers within the composite, if connected.

2.1.3 Declaration of Event Types on Producers and Consumers

Producers can declare the set of event types that they produce in the implementation, component producer or composite producer. Consumers can declare the set of event types that they handle in the implementation, component consumer or composite consumer. Similarly, channels can declare the set of event types that they handle in the composite configuration. It is also possible to declare that a producer, a consumer, or a channel handles any event type.

The value of declaring the events that are produced and consumed by components and channels is that:

- When the event types produced and consumed are explicitly declared, it may be possible to avoid the need for runtime event filters on the consumers, providing an optimized path for the handling of the events.
- Because the channel, producer and consumer declarations can include a list of event types, it is possible to report an error or a warning when a producer or a consumer is connected to a channel, where there is no chance that the produced events will be accepted by the channel or the consumer will ever get any event.

The following always apply:

- A producer SHOULD only produce event type it has declared. [ASM20101]
- An SCA Runtime MAY reject events of a type from a producer which does not declare that it produces events of that type. [ASM20102]

2.2 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 and do not represent any formal graphical notation for SCA.

Figure 2-1 illustrates some of the features of an SCA component:

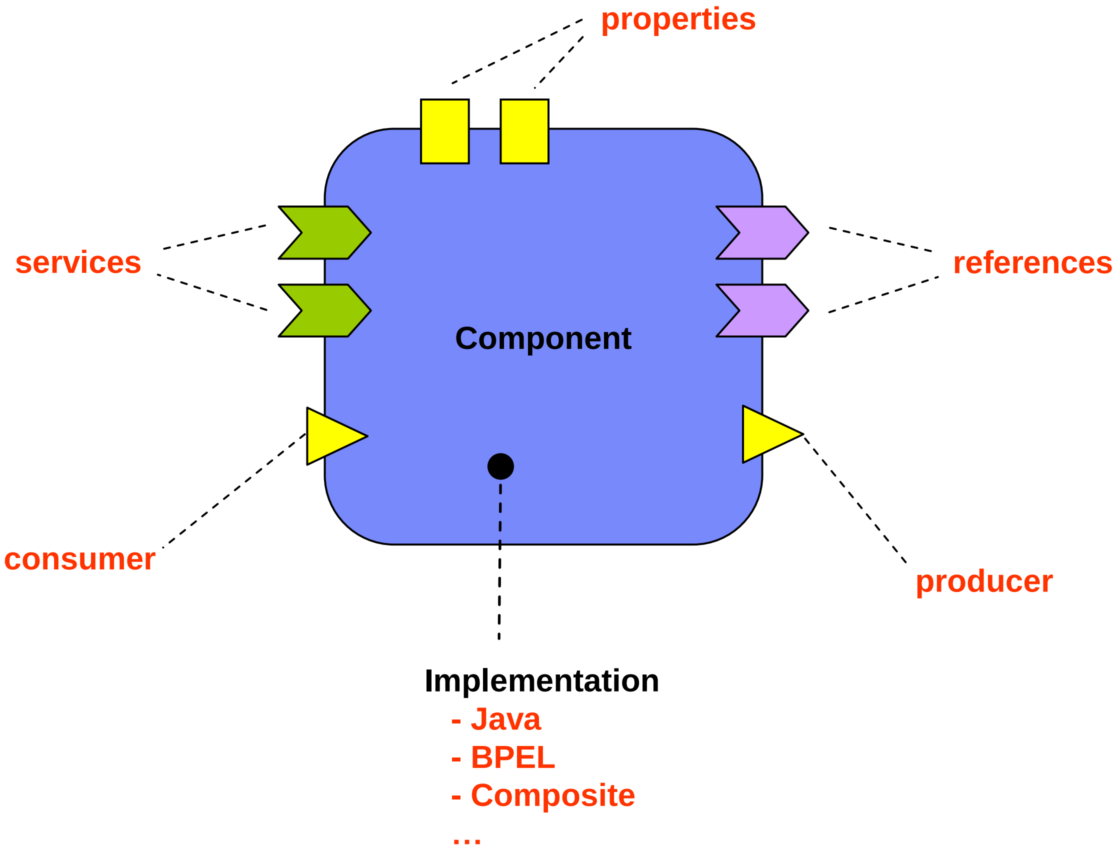
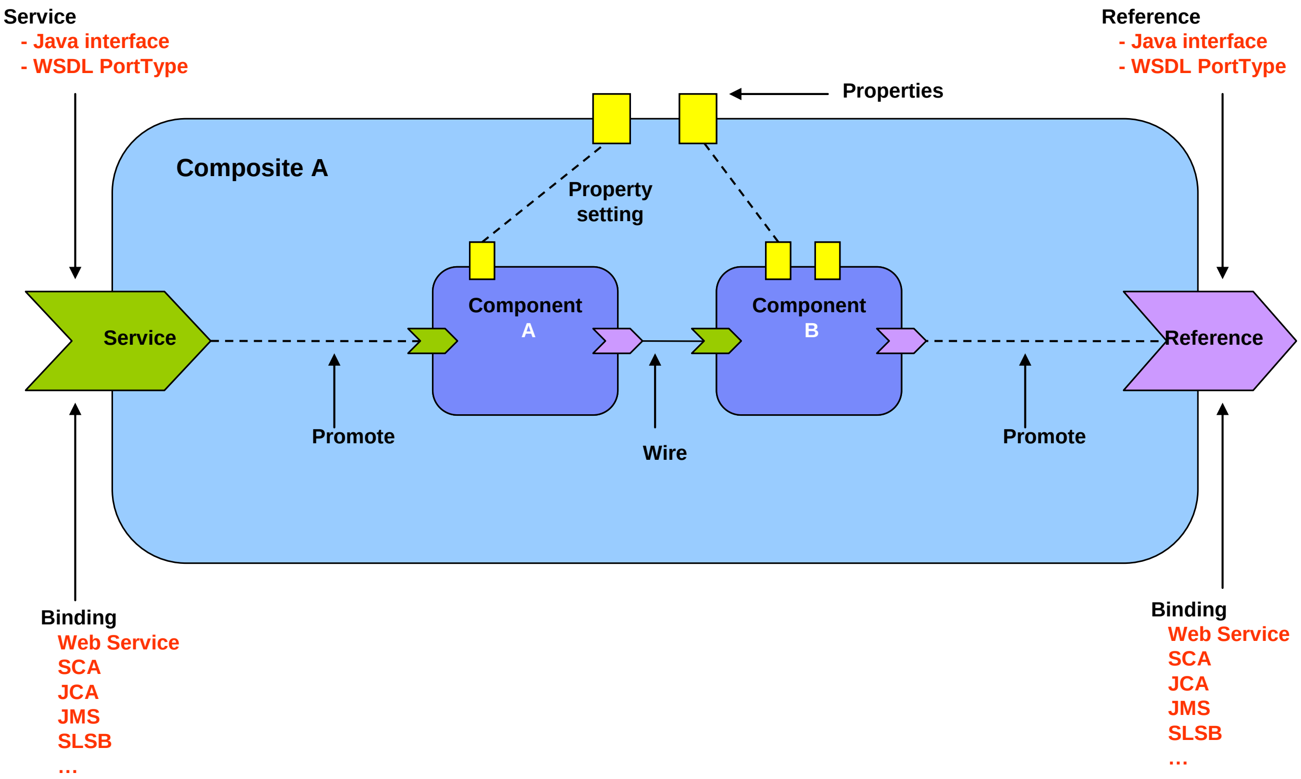


Figure 2-1: SCA Component Diagram

Figure 2-2 illustrates some of the features of a composite assembled using a set of components:



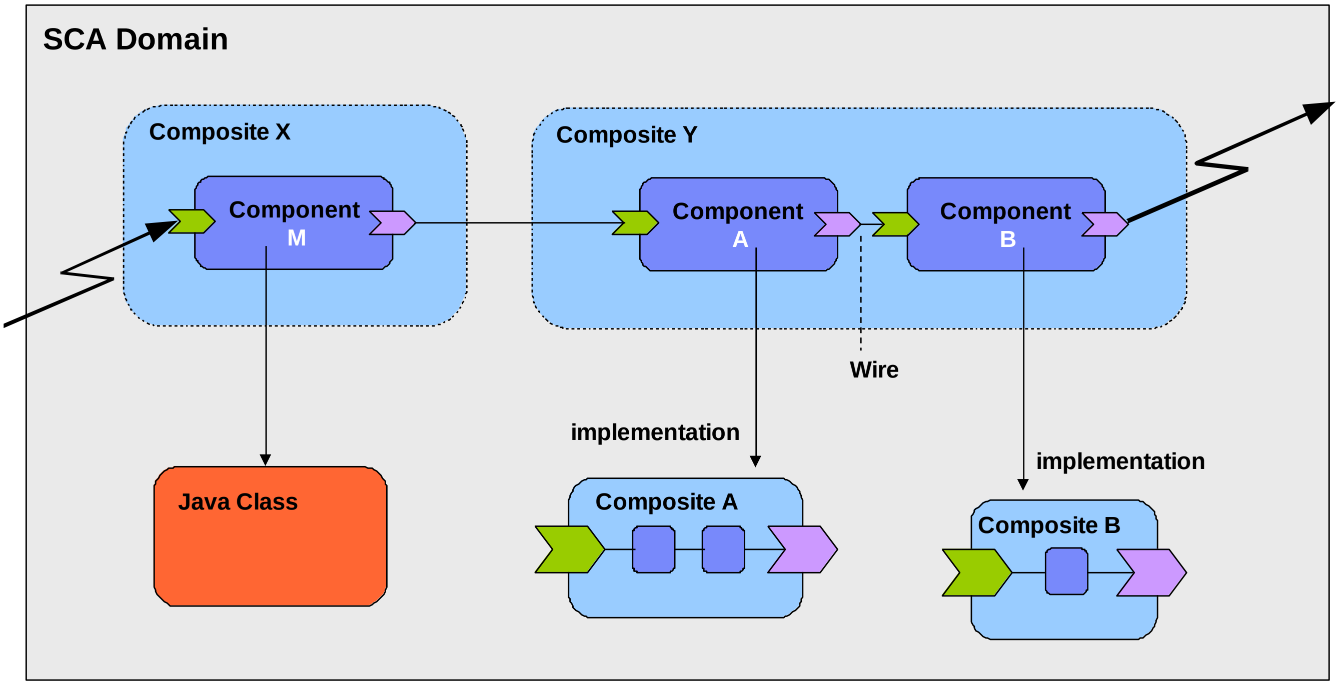
364

365 *Figure 2-2: SCA Composite Diagram*

366

367 Figure 2-3 illustrates an SCA Domain assembled from a series of high-level composites, some of which
368 are in turn implemented by lower-level composites:

369



370

371 *Figure 2-3: SCA Domain Diagram*

372 Figure 2-4 shows and SCA composite involving components that communicate using Pub/Sub:

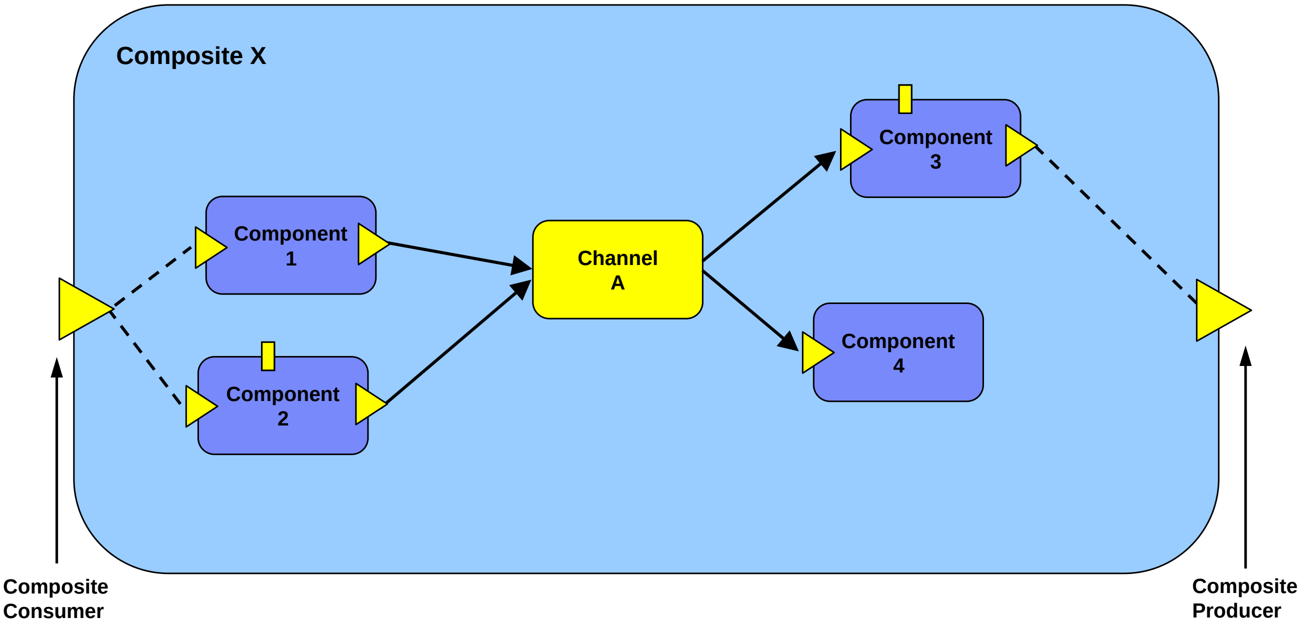


Figure 2-4: SCA Composite Diagram with Pub/Sub

2.3 Pub/Sub Examples

2.3.1 Multiple Producers linked to multiple Consumers via a Channel - within a Composite

This example is of multiple component producers, which send events to multiple component consumers via a Channel, which decouples the producers from the consumers. The assembly is represented by the diagram in Figure 2-5:

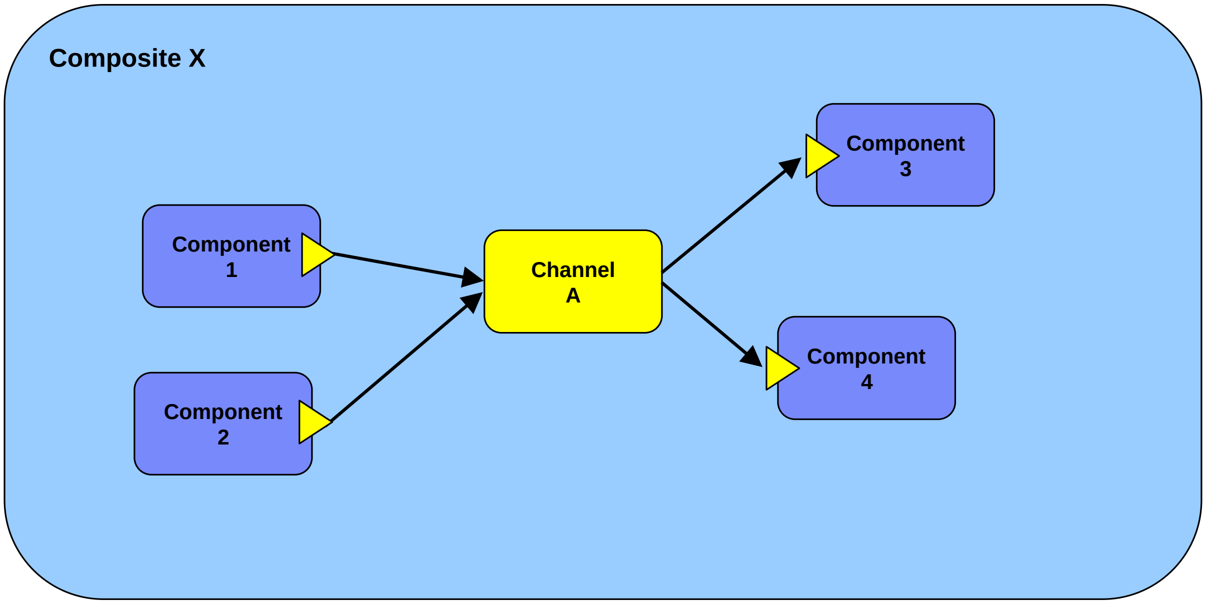


Figure 2-5: Producers Linked to Consumers Via a Local Channel

The corresponding XML for this example follows:

```
<composite name="CompositeX"
   xmlns="http://www.osoa.org/xmlns/sca/1.0"
   targetNamespace="http://example.org/example1">

   <component name="Component1">
         <implementation.java class="org.example.Component1Impl"/>
         <producer name="Foo_Events" target="ChannelA"/>
   </component>

   <component name="Component2">
         <implementation.java class="org.example.Component2Impl"/>
         <producer name="Foo_Events" target="ChannelA"/>
   </component>

   <component name="Component3">
         <implementation.java class="org.example.Component3Impl"/>
         <consumer name="Foo_Handling" source="ChannelA"/>
   </component>

   <component name="Component4">
         <implementation.java class="org.example.Component4Impl"/>
         <consumer name="Foo_Handling" source="ChannelA"/>
   </component>

   <channel name="ChannelA"/>

</composite>
```

In this example, the @target attribute of the producers links them to ChannelA and the @source attribute of the consumers links them to ChannelA. All events from Component1 and Component2 are routed through the ChannelA and are sent to Component3 and Component4.

2.3.2 Producers linked to Consumers via Domain Channels

In this example, component producers of components nested within a domain component transmit events via Domain Channels to component consumers which are also nested below the domain level within a second domain component. This is represented in the Figure 2-6:

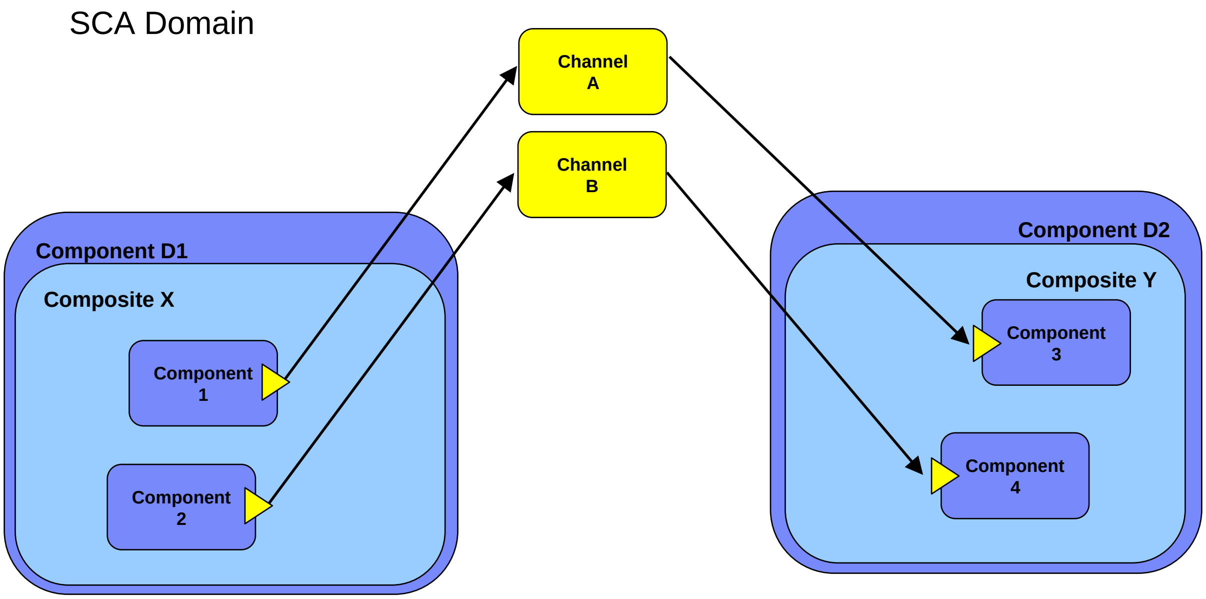


Figure: 2-6: Producers linked to Consumers via Domain Channels

For CompositeX:

```
<composite name="CompositeX"
   xmlns="http://www.osoa.org/xmlns/sca/1.0"
   targetNamespace="http://example.org/example1">

   <component name="Component1">
          <implementation.java class="org.example.Component1Impl"/>
          <producer name="Foo_Events" target="/ChannelA"/>
   </component>

   <component name="Component2">
          <implementation.java class="org.example.Component2Impl"/>
          <producer name="Foo_Events" target="/ChannelB"/>
   </component>

</composite>
```

For CompositeY:

```
<composite name="CompositeY"
   xmlns="http://www.osoa.org/xmlns/sca/1.0"
   targetNamespace="http://example.org/example1">

   <component name="Component3">
          <implementation.java class="org.example.Component3Impl"/>
          <consumer name="Foo_Handling" source="/ChannelA"/>
   </component>

   <component name="Component4">
          <implementation.java class="org.example.Component4Impl"/>
          <consumer name="Foo_Handling" source="/ChannelB"/>
   </component>

</composite>
```

Note the @target and @source attributes of the producers and consumers use the "/<channel-name>" notation to indicate the connection to a channel at the domain level.

The following is an example of one way in which the Channels could be deployed to the Domain:

```
<composite name="ChannelContribution"
   xmlns="http://www.osoa.org/xmlns/sca/1.0"
   targetNamespace="http://example.org/example1">

   <channel name="ChannelA"/>

   <channel name="ChannelB"/>

</composite>
```

The following is an example of two deployment composites that could be used to deploy the two domain-level components (ComponentD1 and ComponentD2):

```
<composite name="ComponentD1Contribution"
   xmlns="http://www.osoa.org/xmlns/sca/1.0"
   targetNamespace="http://example.org/example1"
   xmlns:xmp="http://example.org/example1">

   <component name="ComponentD1">
         <implementation.composite name="xmp:CompositeX"/>
   </component>
</composite>

<composite name="ComponentD2Contribution"
   xmlns="http://www.osoa.org/xmlns/sca/1.0"
   targetNamespace="http://example.org/example1"
   xmlns:xmp="http://example.org/example1">

   <component name="ComponentD2">
         <implementation.composite name="xmp:CompositeY"/>
   </component>

</composite>
```

Note that the domain level components ComponentD1 and ComponentD2 are unable to configure the channels that are used as sources and targets by the components in the lower level composites.

2.3.3 Composite with Promotion of Producers and Consumers

This example shows how a composite can be constructed so that the composite promotes some component cosumers and promotes some component producers. This is represented in Figure 2-7.

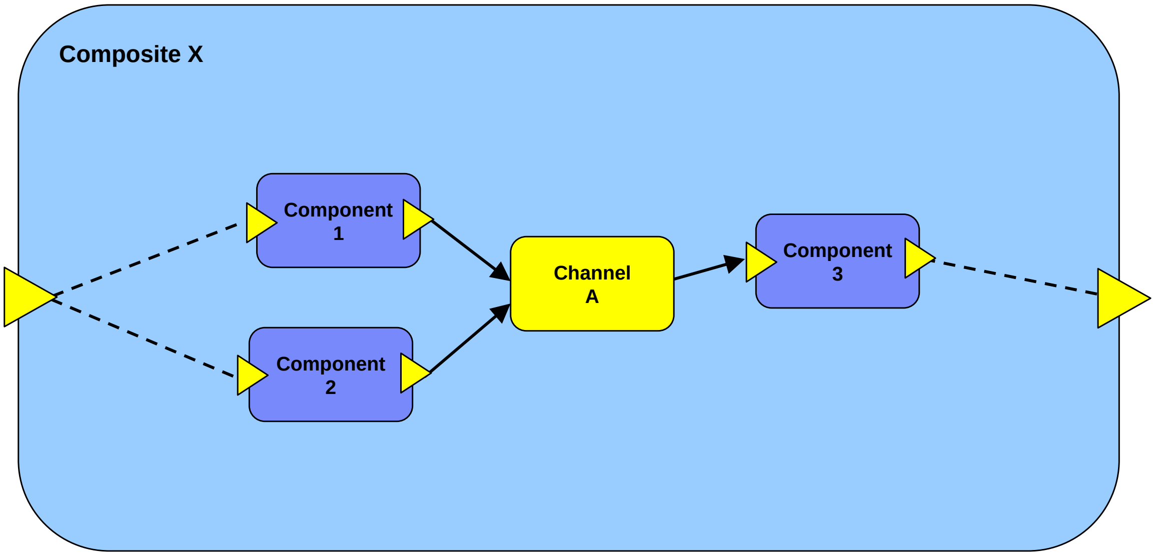


Figure: 2-7 Promotion of Consumers and Producers by a Composite

The corresponding XML for this example follows:

```
<composite name="CompositeX"
   xmlns="http://www.osoa.org/xmlns/sca/1.0"
   targetNamespace="http://example.org/example1">

   <consumer name="Bar_Handling"
            promotes="Component1/BarHandling Component2/Bar_Handling"/>

   <component name="Component1">
         <implementation.java class="org.example.Component1Impl"/>
         <consumer name="Bar_Handling"/>
         <producer name="Foo_Events" target="ChannelA"/>
   </component>

   <component name="Component2">
         <implementation.java class="org.example.Component2Impl"/>
         <consumer name="Bar_Handling"/>
         <producer name="Foo_Events" target="ChannelA"/>
   </component>

   <channel name="ChannelA"/>

   <component name="Component3">
         <implementation.java class="org.example.Component3Impl"/>
         <consumer name="Foo_Handling" source="ChannelA"/>
         <producer name="Special_Events"/>
   </component>

   <producer name="Special_Events" promotes="Component3/Special_Events"/>

</composite>
```

Here, CompositeX has a consumer element named Bar_Handling and producer element named Special_Events. The Bar_Handling consumer promotes the consumers of Component1 and Component2. The Special_Events producer promotes the producer of Component3.

When CompositeX is used as an implementation by a higher-level component, the consumer and producer elements of the composite permit the assembler of the higher level component to control where the events relating to this composite are sent to and received from, through configuration of the higher level component. The Component Type of CompositeX above is as follows:

```
<componentType  xmlns="http://www.osoa.org/xmlns/sca/1.0"

   <consumer name="Bar_Handling" />

   <producer name="Special_Events />

</componentType>
```

3 Implementation and ComponentType

Component ***implementations*** are concrete implementations of business function which provide services and/or which make references to services provided elsewhere. An implementation can also have event producers and consumers. Each producer sends events of one or more event types, while each consumer receives events of one or more event types. Producers and consumers declare the set of event types that they handle through a list of event types. It is also possible to declare that a producer or a consumer handles any event type. 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***, ***consumers***, ***producers***, 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, consumers, producers, and properties that it has and it also might be able to set values for all the characteristics of those services, references, consumers, producers, 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 consumer, the implementation can define event filters, intents, policy sets, bindings
- for a producer, the implementation can define event types, intents, policy sets, 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, consumers, producers, 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, consumers, producers, 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).

3.1 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, consumers to which events are delivered, producers that send out events, and properties that can be set. The settable properties and the settable references to services and the settable consumers and prodcers 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'.

Snippet 3-1 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/200912">

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

</componentType>
```

Snippet 3-1: componentType Pseudo-Schema

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

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

3.1.1 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. Snippet 3-2 shows the componentType pseudo-schema with the pseudo-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/200912" … >

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

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

</componentType>
```

Snippet 3-2: componentType Pseudo-Schema with service Child Element

The ***service*** element has the ***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 : listOfQNames (0..1)*** - a list of policy intents. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.
- ***policySets : listOfQNames (0..1)*** - a list of policy sets. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.

The ***service*** element has the ***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.
- ***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 ***callback*** element is used if the interface has a callback defined, and the callback element has one or more ***binding*** elements as subelements. The ***callback*** and its binding subelements 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.
- ***requires : requires (0..n)*** - A service element has ***zero or more requires subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.
- ***policySetAttachment : policySetAttachment (0..n)*** - A service element has ***zero or more policySetAttachment subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.

3.1.2 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. Snippet 3-3 shows the componentType pseudo-schema with the pseudo-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/200912" … >

   <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 … />
         <binding … />*
         <callback>?
                <binding … />+
         </callback>
         <requires/>*
         <policySetAttachment/>*
   </reference>

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

</componentType>
```

Snippet 3-3: componentType Pseudo-Schema with reference Child Element

The ***reference*** element has the ***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 is 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 (e.g. 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]
- ***requires : listOfQNames (0..1)*** - a list of policy intents. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.
- ***policySets : listOfQNames (0..1)*** - a list of policy sets. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.

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

- ***interface : Interface (1..1)*** - A reference has ***one interface***, which describes the operations used 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.
- ***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.

  When used with a reference element, a binding element specifies an endpoint which is the target of that binding. A reference cannot mix the use of endpoints specified via binding elements with target endpoints specified via the @target attribute. If the @target attribute is set, the reference cannot also have binding subelements. If binding elements with endpoints are specified, each endpoint uses the binding type of the binding element in which it is defined.
- ***callback (0..1) / binding : Binding (1..n)*** - al ***callback*** element is used if the interface has a callback defined and the callback element has one or more ***binding*** elements as subelements. The ***callback*** and its binding subelements 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.
- ***requires : requires (0..n)*** - A service element has ***zero or more requires subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.
- ***policySetAttachment : policySetAttachment (0..n)*** - A service element has ***zero or more policySetAttachment subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.

For a full description of the setting of target service(s) for a reference, see the section "Specifying the Target Service(s) for a Reference".

3.1.3 Consumer

A Consumer is represented by a ***consumer element*** which is a child of the componentType element. There can be ***zero or more*** consumer elements in a componentType. Snippet 3-4 shows the componentType pseudo-schema with the pseudo-schema for a consumer child element:

```
<!-- Component type consumer schema snippet -->
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912" … >

   <service … />*
   <reference … />*

   <consumer name="xs:NCName"
         requires="list of xs:QName"? policySets="list of xs:QName"?>*
         <filters … />?
         <requires/>*
         <policySetAttachment/>*
   </consumer>

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

</componentType>
```

Snippet 3-4: componentType Pseudo-Schema with consumer Child Element

The ***consumer*** element has the ***attributes***:

- ***name : NCName (1..1)*** - the name of the consumer. The @name attribute of a <consumer/> child element of a <componentType/> MUST be unique amongst the consumer elements of that <componentType/>. [ASM40101]
- ***requires : listOfQNames (0..1)*** - a list of policy intents. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.

- ***policySets : listOfQNames (0..1)*** - a list of policy sets. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.

The ***consumer*** element has the ***child elements***:

- ***filter : Filter (0..1)*** - A consumer has ***zero or one filter***, which specify the events of interest for the consumer. For details on the filter element see the Filters: Selecting Subsets of Events section.
- ***requires : requires (0..n)*** - A service element has ***zero or more requires subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.
- ***policySetAttachment : policySetAttachment (0..n)*** - A service element has ***zero or more policySetAttachment subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.

In the SCA Assembly Model an implementation-specific artifact, for example a Java Class, can manifest itself as both an SCA service and an SCA consumer in the component Type of that implementation. This dual manifestation of an artifact is allowed to facilitate those usecases where the distinction between the two is not relevant to the implementation code. In such a case, it is the assembler's responsibility to decide whether the artifact would receive events from a channel or whether it would receive method invocations from a reference. It is up to each implementation type to decide whether such a feature is supported and how the distinction between dual, consumer-only, and service-only manifestation of an artifact is made. When implementation has an artifact that has dual manifestation, the effective component type of that implementation contains a component type service corresponding to the artifact and a consumer corresponding to the same artifact. The name of the component type service and the name of the component type consumer associated with the artifact are the same. In such a case, an assembler cannot use both the service and the consumer.

For artifacts that have service/consumer dual manifestation, the service interface MUST

- be non-bidirectional
- have only one-way operations

[ASM40102]

```
Note: For this dual nature to work the TC has to define a WSDL-to-EDL mapping.
Note: The F2F notes do not contain a decision about dual nature
service/consumer … and recollections of various ppl who attended the f2f vary.
```

3.1.4 Producer

A Producer is represented by a ***producer element*** which is a child of the componentType element. There can be ***zero or more*** producer elements in a componentType. Snippet 3-5 shows the componentType pseudo-schema with the pseudo-schema for a producer child element:

```
<!-- Component type consumer schema snippet -->
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912" … >

   <service … />*
   <reference … />*
   <consumer … />*

   <producer name="xs:NCName"
          requires="list of xs:QName"? policySets="list of xs:QName"?>*
          <eventType … />?
         <requires/>*
         <policySetAttachment/>*
   </producer>

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

</componentType>
```

Snippet 3-5: componentType Pseudo-Schema with producer Child Element

The ***producer*** element has the ***attributes***:

- ***name : NCName (1..1)*** - the name of the producer. The @name attribute of a <producer/> child element of a <componentType/> MUST be unique amongst the producer elements of that <componentType/>. [ASM40103]
- ***requires : listOfQNames (0..1)*** - a list of policy intents. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.
- ***policySets : listOfQNames (0..1)*** - a list of policy sets. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.

The ***producer*** element has the ***child elements***:

- ***eventType : EventType (0..1)*** - A producer has ***zero or one eventType*** child subelement. See the Section Use of <eventType> on a Producer.
- ***requires : requires (0..n)*** - A service element has ***zero or more requires subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.
- ***policySetAttachment : policySetAttachment (0..n)*** - A service element has ***zero or more policySetAttachment subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.

In the SCA Assembly Model an implementation-specific artifact, for example a Java Class instance variable, can manifest itself as both an SCA reference and an SCA producer in the component Type of that implementation. This dual manifestation of an artifact is allowed to facilitate those usecases where the distinction between the two is not relevant to the implementation code. In such a case, it is the assembler's responsibility to decide whether the artifact would send events to a channel or whether the it would invoke methods on a service. It is up to each implementation type to decide whether such a feature is supported and how the distinction between dual, producer-only, and reference-only nature of an artifact is made. When implementation has an artifact that has dual manifestation, the effective component type of that implementation contains a component type reference corresponding to the artifact and a producer corresponding to the same artifact. The name of the component type reference and the name of the component type producer associated with the artifact are the same. In such a case, an assembler cannot use both the reference and the producer.

For artifacts that have reference/producer dual manifestation, the reference interface MUST

- be non-bidirectional
- have only one-way operations

[ASM40104]

```
Note: For this dual nature to work the TC has to define a WSDL-to-EDL mapping.
```

3.1.5 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. Snippet 3-6 shows the componentType pseudo-schema with the pseudo-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/200912" … >

   <service … />*
   <reference … >*
   <consumer … />*
   <producer … />*

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

   <implementation … />?

</componentType>
```

Snippet 3-6: componentType Pseudo-Schema with property Child Element

The ***property*** element has the ***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]

  A single property element MUST NOT contain both a @type attribute and an @element attribute. [ASM40010]
- ***many : boolean (0..1)*** - 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 needs to be supplied by the component that uses the implementation. Default value is "false". When the componentType has @mustSupply="true" for a property element, a component using the implementation MUST supply a value for the property since the implementation has no default value for the property. [ASM40011] If the implementation has a default-property-value then @mustSupply="false" is appropriate, since the implication of a default value is that it is used when a value is not supplied by the using component.
- ***file : anyURI (0..1)*** - a dereferencable URI to a file containing a value for the property. The value of the property @file attribute MUST be a dereferencable URI to a file containing the value for the property. [ASM40012] The URI can be an absolute URI or a relative URI. For a relative URI, it is taken relative to the base of the contribution containing the implementation. For a description of the format of the file, see the section on Property Value File Format.

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 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]

3.1.6 Implementation

Implementation represents characteristics inherent to the implementation itself, in particular intents and policies. See the Policy Framework specification [SCA-POLICY] for a description of intents and policies. Snippet 3-7 shows the componentType pseudo-schema with the pseudo-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/200912" … >

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

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

</componentType>
```

Snippet 3-7: componentType Pseudo-Schema with implementation Child Element

The ***implementation*** element has the ***attributes***:

- ***requires : listOfQNames (0..1)*** - a list of policy intents. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.
- ***policySets : listOfQNames (0..1)*** - a list of policy sets. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.

The ***implementation*** element has the ***subelements***:

- ***requires : requires (0..n)*** - A service element has ***zero or more requires subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.
- ***policySetAttachment : policySetAttachment (0..n)*** - A service element has ***zero or more policySetAttachment subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.

3.2 Example ComponentType

Snippet 3-8 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/200912
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:storm="http://example.org/storm">

   <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>

   <consumer name="stormAlertConsumer">
         <filters>
               <eventType.sca qnames="storm:SouthAmerica" />
         </filters>
   </consumer>

   <producer name="stormAlertProducer">
         <eventType.sca qnames="storm:NorthAmerica" />
   </producer>

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

</componentType>
```

Snippet 3-8: Example componentType

3.3 Example Implementation

Snippet 3-9 and Snippet 3-10 are an example implementation, written in Java.

AccountServiceImpl implements the ***AccountService*** interface, which is defined via a Java interface (TODO: update when the Java CI adds support for events):

```
package services.account;

@Remotable
public interface AccountService {

   AccountReport getAccountReport(String customerID);
}
```

Snippet 3-9: Example Interface in Java

Snippet 3-10 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, @Reference and @Service annotations:

```
package services.account;

import java.util.List;

import commonj.sdo.DataFactory;

import org.oasisopen.sca.annotation.Property;
import org.oasisopen.sca.annotation.Reference;
import org.oasisopen.sca.annotation.Service;

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

@Service(AccountService.class)
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;
   }
}
```

Snippet 3-10: Example Component Implementation in Java

Snippet 3-11 is the SCA componentType definition for the AccountServiceImpl, derived by introspection of the code above:

```
<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
                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"/>

</componentType>
```

Snippet 3-11: Example componentType for Implementation in Snippet 3-10

Note that the componentType property element for "currency" has no default value declared, despite the code containing an initializer for the property field setting it to "USD". This is because the initializer cannot be introspected at runtime and the value cannot be extracted.

For full details about Java implementations, see the Java Component Implementation Specification [SCA-Java]. Other implementation types have their own specification documents.

4 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 and/or events. 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 a file with a ***.composite*** extension. 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. Snippet 4-1 shows the composite pseudo-schema with the pseudo-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/200912" … >
  …
  <component name="xs:NCName" autowire="xs:boolean"?
             requires="list of xs:QName"? policySets="list of xs:QName"?>*
     <implementation … />?
     <service … />*
     <reference … />*
     <consumer … />*
     <producer … />*
     <property … />*
     <requires/>*
     <policySetAttachment/>*
  </component>
  …
</composite>
```

Snippet 4-1: composite Pseudo-Schema with component Child Element

The ***component*** element has the ***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 are autowired, as described in the Autowire section. Default is false.
- ***requires : listOfQNames (0..1)*** – a list of policy intents. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.
- ***policySets : listOfQNames (0..1)*** – a list of policy sets. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.

The ***component*** element has the ***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.
- consumer: Consumer (0..n) – see component consumer section.
- producer: Producer (0..n) – see component producer section.
- ***property : ComponentProperty (0..n)*** – see component property section.

- ***requires : requires (0..n)*** - A service element has ***zero or more requires subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.
- ***policySetAttachment : policySetAttachment (0..n)*** - A service element has ***zero or more policySetAttachment subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.

4.1 Implementation

A component element has ***one implementation element*** as its child, which points to the implementation used by the component.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Component Implementation schema snippet -->
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912" … >
  …
  <component … >*
     <implementation requires="list of xs:QName"?
        policySets="list of xs:QName"?>
        <requires/>*
        <policySetAttachment/>*
     </implementation>
     <service … />*
     <reference … />*
     <consumer … />*
     <producer … />*
     <property … />*
  </component>
  …
</composite>
```

Snippet 4-2: component Pseudo-Schema with implementation Child Element

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.

Snippet 4-3 – Snippet 4-5 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"/>
```

Snippet 4-3: Example implementation.java Element

```
<implementation.bpel process="ans:MoneyTransferProcess"/>
```

Snippet 4-4: Example implementation.bpel Element

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

Snippet 4-5: Example implementation.composite Element

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.

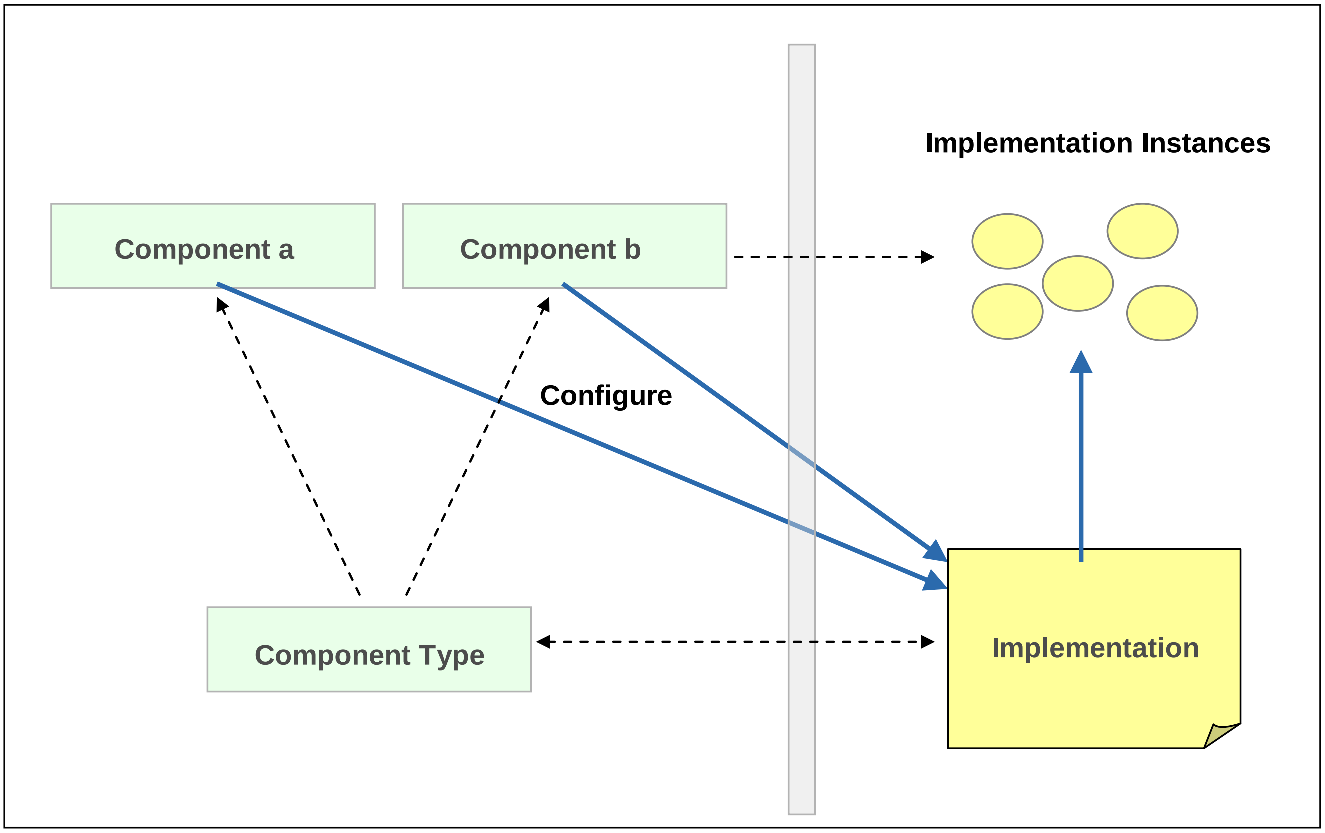


Figure 4-1: Relationship of Component and Implementation

4.2 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. Snippet 4-6 shows the component pseudo-schema with the pseudo-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/200912" … >
   …
   <component … >*
      <implementation … />
      <service name="xs:NCName" requires="list of xs:QName"?
         policySets="list of xs:QName"?>*
         <interface … />?
         <binding … />*
         <callback>?
            <binding … />+
         </callback>
         <requires/>*
         <policySetAttachment/>*
      </service>
      <reference … />*
      <consumer … />*
      <producer … />*
      <property … />*
   </component>
   …
</composite>
```

Snippet 4-6: component Pseudo-Schema with service Child Element

The ***component service*** element has the ***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 : listOfQNames (0..1)*** – a list of policy intents. See the Policy Framework specification [SCA-POLICY] 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 : listOfQNames (0..1)*** – a list of policy sets. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.

The ***component service*** element has the ***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 an interface is declared for a component service, the interface MUST provide a compatible subset of the interface declared for the equivalent service in the componentType of the implementation [ASM50004] For details on the interface element see the Interface section.
- ***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 [SCA-POLICY].
- ***callback (0..1) / binding : Binding (1..n)*** - A ***callback*** element is used if the interface has a callback defined and the callback element has one or more ***binding*** elements as subelements. The ***callback*** and its binding subelements 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.
- ***requires : requires (0..n)*** - A service element has ***zero or more requires subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.
- ***policySetAttachment : policySetAttachment (0..n)*** - A service element has ***zero or more policySetAttachment subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.

4.3 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. Snippet 4-7 shows the component pseudo-schema with the pseudo-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/200912" … >
   …
   <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"?
         nonOverridable="xs:boolean"
         wiredByImpl="xs:boolean"? requires="list of xs:QName"?
         policySets="list of xs:QName"?>*
         <interface … />?
         <binding uri="xs:anyURI"? requires="list of xs:QName"?
            policySets="list of xs:QName"?/>*
         <callback>?
            <binding … />+
         </callback>
         <requires/>*
         <policySetAttachment/>*
      </reference>
      <consumer … />*
      <producer … />*
      <property … />*
   </component>
   …
</composite>
```

Snippet 4-7: component Pseudo-Schema with reference Child Element

The ***component reference*** element has the ***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 is autowired, as described in the Autowire section. The default value of the @autowire attribute MUST be the value of the @autowire attribute on the component containing the reference, if present, or else the value of the @autowire attribute of the composite containing the component, if present, and if neither is present, then it is "false". [ASM50043]
- ***requires : listOfQNames (0..1)*** – a list of policy intents. See the Policy Framework specification [SCA-POLICY] 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 : listOfQNames (0..1)*** – a list of policy sets. See the Policy Framework specification [SCA-POLICY] 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 (e.g. 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]
- ***nonOverridable : boolean (0..1)*** - a boolean value, "false" by default, which indicates whether this component reference can have its targets overridden by a composite reference which promotes the component reference.
  If @nonOverridable==false, if any target(s) are configured onto the composite references which promote the component reference, then those targets ***replace*** all the targets explicitly declared on the component reference for any value of @multiplicity on the component reference. If no targets are defined on any of the composite references which promote the component reference, then any targets explicitly declared on the component reference are used. This means in effect that any targets declared on the component reference act as default targets for that reference.

  If a component reference has @multiplicity 0..1 or 1..1 and @nonOverridable==true, then the component reference MUST NOT be promoted by any composite reference. [ASM50042]

  If @nonOverridable==true, and the component reference @multiplicity is 0..n or 1..n, any targets configured onto the composite references which promote the component reference are added to any references declared on the component reference - that is, the targets are additive.

The component reference element has the child elements:

- ***interface : Interface (0..1)*** - A reference has ***zero or one interface***, which describes the operations of 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, the interface MUST provide a compatible superset of the interface declared for the equivalent reference in the componentType of the implementation. [ASM50011] For details on the interface element see the Interface section.
- ***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. 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] It is valid for there to be no binding elements on the component reference and none on the reference in the componentType - the binding used for such a reference is determined by the target service. See the section on the bindings of component services for a description of how the binding(s) applying to a service are determined.

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 [SCA-POLICY].

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 "Specifying the Target Service(s) for a Reference"

- ***callback (0..1) / binding : Binding (1..n)*** - A ***callback*** element used if the interface has a callback defined and the callback element has one or more ***binding*** elements as subelements. The ***callback*** and its binding subelements 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.
- ***requires : requires (0..n)*** - A service element has ***zero or more requires subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.
- ***policySetAttachment : policySetAttachment (0..n)*** - A service element has ***zero or more policySetAttachment subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.

4.3.1 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 uses more than a simple URI for the address of a target service. In cases where a reference element has a binding subelement that uses more than simple URI, the @uri attribute of the binding element MUST NOT be used to identify the target service - in this case 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]

4.3.1.1 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 MUST have no more than one target service defined. [ASM50039]
- A reference with multiplicity 1..1 MUST have exactly one target service defined. [ASM50040]
- A reference with multiplicity 1..n MUST have at least one target service defined. [ASM50041]
- A reference with multiplicity 0..n can have any number of target services defined.

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 raise an error no later than when the reference is invoked by the component implementation. [ASM50022]

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.

A contrasting example is a component deployed to the SCA Domain. At the Domain level, the target of a wire, or even the wire itself, can form part of a separate deployed contribution and as a result these can 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]

4.4 Consumer

The component element can have ***zero or more consumer elements*** as children, which are used to configure the consumers of the component. The consumers that can be configured are defined by the implementation. Snippet 4-8 shows the component pseudo-schema with the pseudo-schema for a consumer child element.

```
<!-- Component Consumer schema snippet -->
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912" … >
   …
   <component … >*
      <implementation … />
      <service … />*
      <reference … />*
      <consumer name="xs:NCName"
                requires="list of xs:QName"?
                policySets="list of xs:QName"?
                source="list of xs:anyURI"?>
```

```
         <filters/>?
         <requires/>*
         <policySetAttachment/>*
      </consumer>*
      <producer … />*
      <property … />*
   </component>
   …
</composite>
```

Snippet 4-8: Component Pseudo-Schema with consumer Child Element

The consumer element has the following attributes:

- ***name: NCName (1..1)*** – the name of the consumer. The @name attribute of a consumer element of a <component/> MUST be unique amongst the consumer elements of that <component/>. [ASM50101] The @name attribute of a consumer element of a <component/> MUST match the @name attribute of a consumer element of the componentType of the <implementation/> child element of the component. [ASM50102]
- ***requires : listOfQNames (0..1)*** – a list of policy intents. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.
  Note: The effective set of policy intents for the consumer consists of any intents explicitly stated in this @requires attribute, combined with any intents specified for the service by the implementation.
- ***policySets : listOfQNames (0..1)*** – a list of policy sets. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.
- ***source: listOfAnyURIs (0..1)*** – a list of one or more of event sources such as the URI of a channel. The form of the URI for a channel is described in section The URI of a Channel.

The consumer element has the following child elements:

- ***filters: Filters (0..1) –*** filter elements. See the section Filters: Selecting Subsets of Events for a detailed description of filters.
- ***requires : requires (0..n)*** - A service element has ***zero or more requires subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.
- ***policySetAttachment : policySetAttachment (0..n)*** - A service element has ***zero or more policySetAttachment subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.

The consumer can receive events from all of the event sources identified in the @source attribute. It is valid to specify no sources (ie the consumer is "unconnected"). If the consumer is unconnected, no events are received.If the name of the consumer is the same as a service within the same component, then both the consumer and the service MUST NOT be connected.

4.5 Producer

The component element can have ***zero or more producer elements*** as children, which are used to configure the producers of the component. The producers that can be configured are defined by the implementation. Snippet 4-9 shows the component pseudo-schema with the pseudo-schema for a producer child element.

```
<!-- Component Consumer schema snippet -->
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912" … >
   …
   <component … >*
      <implementation … />
      <service … />*
      <reference … />*
      <consumer … />*
      <producer name="xs:NCName"
                requires="list of xs:QName"?
                policySets="list of xs:QName"?
                target="list of xs:anyURI"?>
         <eventType … />?
         <requires/>*
         <policySetAttachment/>*
      </producer>*
      <property … />*
   </component>
   …
</composite>
```

Snippet 4-9: Component Pseudo-Schema with producer Child Element

The producer element has the following attributes:

- ***name: NCName (1..1)*** – the name of the producer. The @name attribute of a producer element of a <component/> MUST be unique amongst the producer elements of that <component/>. [ASM50103] The @name attribute of a producer element of a <component/> MUST match the @name attribute of a producer element of the componentType of the <implementation/> child element of the component. [ASM50104]
- ***requires : listOfQNames (0..1)*** – a list of policy intents. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.
  Note: The effective set of policy intents for the producer consists of any intents explicitly stated in this @requires attribute, combined with any intents specified for the service by the implementation.
- ***policySets : listOfQNames (0..1)*** – a list of policy sets. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.
- ***target: listOfURIs (0..1)*** – a list of one or more of targets to which events are sent, such as the URI of a channel. Where multiple targets are identified, all the messages emitted by the producer are sent to each target. The form of the URI for a channel is described in section The URI of a Channel.

The producer element has the following child elements:

- ***eventType : EventType (0..1)*** - A producer has ***zero or one eventType*** child subelement. See Section Use of <eventType> on a Producer.
- ***requires : requires (0..n)*** - A service element has ***zero or more requires subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.
- ***policySetAttachment : policySetAttachment (0..n)*** - A service element has ***zero or more policySetAttachment subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.

Events produced by the producer are sent to all the targets identified in the @target attribute. It is valid to specify no targets (ie the producer is "unconnected") - in this case events produced are discarded.

If the name of the producer is the same as a reference within the same component, then both the producer and the reference MUST NOT be connected.

4.6 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" />
```

Snippet 4-10: Example property using @value attribute

- As a value, supplied as the content of the ***value*** subelement(s) 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>
```

Snippet 4-11: Example property with a Simple Type Containing a Single Value

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

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

Snippet 4-12: Example property with a Simple Type Containing Multiple Values

  – 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>
```

Snippet 4-13: Example property with a Complex Type Containing a Single Value

  – 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>
```

Snippet 4-14: Example property with a Complex Type Containing Multiple Values

- 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>
```

Snippet 4-15: Example property with a Global Element Declaration Containing a Single Value

  – 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>
```

Snippet 4-16: Example property with a Global Element Declaration Containing Multiple Values

- 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, all the values MUST be contained within a single property element. [ASM50044]

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 for the property element of a component MUST be compatible with the type of the property with the same @name declared in the component type of the implementation used by the component. If no type is declared in the component property element, the type of the property declared in the componentType of the implementation MUST be used. [ASM50036]

The meaning of "compatible" for property types is defined in the section Property Type Compatibility.

Snippet 4-17 shows the component pseudo-schema with the pseudo-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/200912" … >
   …
   <component … >*
          <implementation … />?
          <service … />*
          <reference … />*
          <property name="xs:NCName"
                    (type="xs:QName" | element="xs:QName")?
                    many="xs:boolean"?
                    source="xs:string"? file="xs:anyURI"?
                    value="xs:string"?>*
                 [<value>+ | xs:any+ ]?
          </property>
   </component>
   …
</composite>
```

Snippet 4.17: component Pseudo-Schema with property Child Element

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

- ***name : NCName (1..1)*** – the name of the property. The @name attribute of a property element of a <component/> MUST be unique amongst the property elements of that <component/>. [ASM50031] The @name attribute of a property element of a <component/> MUST match the @name attribute of a property element of the componentType of the <implementation/> child element of the component. [ASM50037]
- 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

  A single property element MUST NOT contain both a @type attribute and an @element attribute. [ASM50035]
- ***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. The value of the component property @file attribute MUST be a dereferencable URI to a file containing the value for the property. [ASM50045] The URI can be an absolute URI or a relative URI. For a relative URI, it is taken relative to the base of the contribution containing the composite in which the component is declared. For a description of the format of the file, see the section on Property Value File Format.
- ***many : boolean (0..1)*** – whether the property is single-valued (false) or multi-valued (true). Overrides the many specified for this property in the componentType of 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.

The ***component property*** element has the ***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]

4.6.1 Property Type Compatibility

There are a number of situations where the declared type of a property element is matched with the declared type of another property element. These situations include:

- Where a component <property/> sets a value for a property of an implementation, as declared in the componentType of the implementation
- Where a component <property/> gets its value from the value of a composite <property/> by means of its @source attribute. This situation can also involve the @source attribute referencing a subelement of the composite <property/> value, in which case it is the type of the subelement which must be matched with the type of the component <property/>
- Where the componentType of a composite used as an implementation is calculated and componentType <property/> elements are created for each composite <property/>

In these cases where the types of two property elements are matched, the types declared for the two <property/> elements MUST be compatible [ASM50038]

Two property types are compatible if they have the same XSD type (where declared as XSD types) or the same XSD global element (where declared as XSD global elements). For cases where the type of a property is declared using a different type system (eg Java), then the type of the property is mapped to XSD using the mapping rules defined by the appropriate implementation type specification

4.6.2 Property Value File Format

The format of the file which is referenced by the @file attribute of a component property or a componentType property is that it is an XML document which MUST contain an sca:values element which in turn contains one of:

- a set of one or more <sca:value/> elements each containing a simple string - where the property type is a simple XML type
- a set of one or more <sca:value/> elements or a set of one or more global elements - where the property type is a complex XML type

[ASM50046]

```
<?xml version="1.0" encoding="UTF-8"?>
<values>
   <value>MyValue</value>
</values>
```

Snippet 4-17: Property Value File Content for simple property type

```
<?xml version="1.0" encoding="UTF-8"?>
<values>
   <foo:fooElement>
      <foo:a>AValue</foo:a>
      <foo:b>InterestingURI</foo:b>
   </foo:fooElement>
</values/>
```

Snippet 4-18: Property Value File Content for a complex property type

4.7 Example Component

Figure 4-2 shows the ***component symbol*** that is used to represent a component in an assembly diagram.

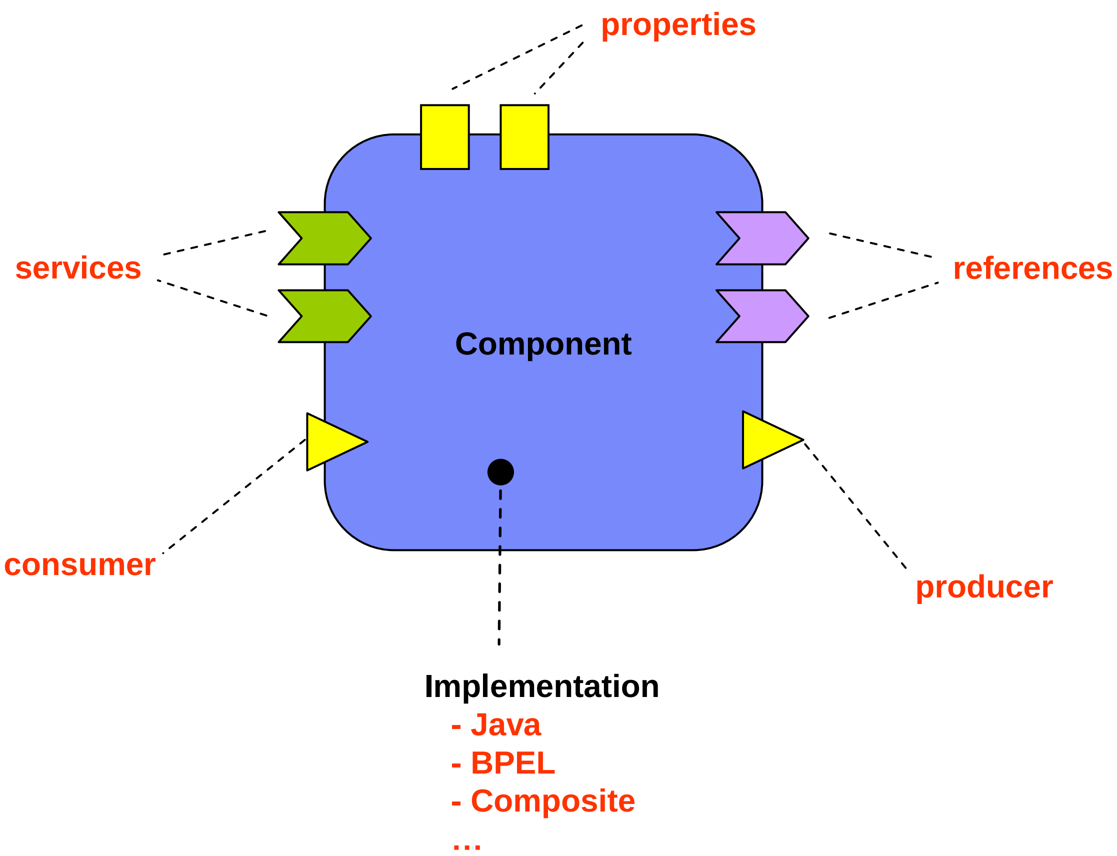


Figure 4-2: Component symbol

Figure 4-3 shows the assembly diagram for the MyValueComposite containing the MyValueServiceComponent.

(TODO: modify the figure/example to include pub/sub/channels)

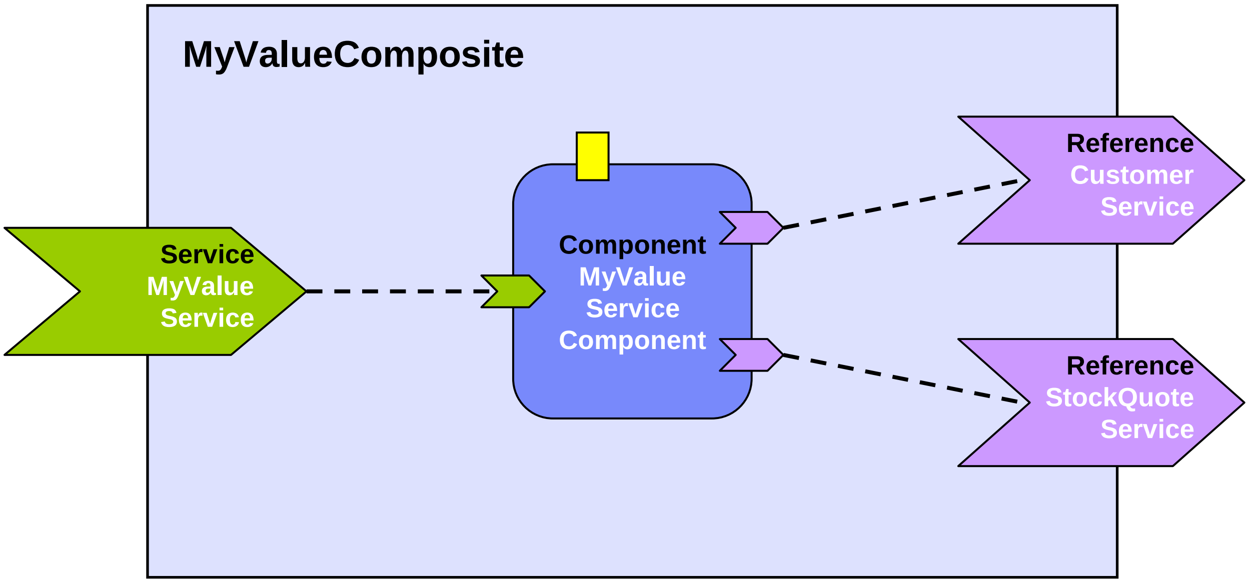


Figure 4-3: Assembly diagram for MyValueComposite

Snippet 4-19 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/200912"
                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>
```

Snippet 4-19: Example 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.

Snippet 4-20 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/200912"
                targetNamespace="http://foo.com"
                name="MyValueComposite" >

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

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

   ...

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

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

</composite>
```

Snippet 4-20: Example composite with Multi-Valued property and reference

….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.

5 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, channels, consumers, producers, 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 components and wires 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. Snippet 5-1 shows the pseudo-schema for the composite element:

```
<?xml version="1.0" encoding="ASCII"?>
<!-- Composite schema snippet -->
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
         targetNamespace="xs:anyURI"
         name="xs:NCName" local="xs:boolean"?
         autowire="xs:boolean"?
         requires="list of xs:QName"? policySets="list of xs:QName"?>

   <include … />*

   <requires/>*
   <policySetAttachment/>*

   <service … />*
   <reference … />*

   <channel … />*
   <consumer … />*
   <producer … />*

   <property … />*

   <component … />*

   <wire … />*

</composite>
```

Snippet 5-1: composite Pseduo-Schema

The ***composite*** element has the ***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 attribute value MUST be unique within the namespace of the composite. [ASM60001]
- ***targetNamespace : anyURI (1..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 are autowired, as described in the Autowire section. Default is false.
- ***requires : listOfQNames (0..1)*** – a list of policy intents. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.
- ***policySets : listOfQNames (0..1)*** – a list of policy sets. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.

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

- ***service : CompositeService (0..n)*** – see composite service section.
- ***reference : CompositeReference (0..n)*** – see composite reference section.
- ***channel: Channel (0..n)*** – see channel section.
- ***consumer : CompositeConsumer (0..n)*** – see composite consumer section.
- ***producer: CompositeProducer (0..n)*** – see composite producer 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
- ***requires : requires (0..n)*** - A service element has ***zero or more requires subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.
- ***policySetAttachment : policySetAttachment (0..n)*** - A service element has ***zero or more policySetAttachment subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.

Components contain configured implementations which hold the business logic of the composite. The components offer services and use references to other services and they send out events via producers and receive events through consumers.

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 each of the component references has an interface that is a compatible subset of the interface on the composite reference. 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.

Channels within the composite represent intermediaries transmitting events from producers to consumers entirely within the composite. Composite consumers define public locations where events are received from outside the composite. Composite producers represent places where the composite as a whole sends out events. Composite consumers involve the ***promotion*** of one or more contained component consumers. Composite producers involve the ***promotion*** of one or more contained component producers.

Component producers can be promoted to composite producers and can be configured to send events to other targets at the same time. Similarly, component consumers can be promoted to composite consumers and can be configured to receive events from other sources at the same time.

5.1 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. Snippet 5-2 shows the composite pseudo-schema with 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/200912" … >
   …
   <service name="xs:NCName" promote="xs:anyURI"
      requires="list of xs:QName"? policySets="list of xs:QName"?>*
      <interface … />?
      <binding … />*
      <callback>?
         <binding … />+
      </callback>
      <requires/>*
      <policySetAttachment/>*
   </service>
   …
</composite>
```

Snippet 5-2: composite Pseudo-Schema with service Child Element

The ***composite service*** element has the ***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 can be omitted 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] <include/> processing MUST take place before the processing of the @promote attribute of a composite service is performed. [ASM60038]
- ***requires : listOfQNames (0..1)*** – a list of policy intents. See the Policy Framework specification [SCA-POLICY] for a description of this attribute. Specified intents add to or further qualify the required intents defined by the promoted component service.

- ***policySets : listOfQNames (0..1)*** – a list of policy sets. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.

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

- ***interface : Interface (0..1)*** - an interface which decribes the operations provided by the composite service. 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. [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.
- ***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 ***callback*** element is used if the interface has a callback defined and the callback has one or more ***binding*** elements as subelements. The ***callback*** and its binding subelements are specified if there is a need to have binding details used to handle callbacks. Callback binding elements attached to the composite service override any callback binding elements defined on the promoted component service. If the callback element is not present on the composite service, any callback binding elements on the promoted service are used. If the callback element is not present at all, the behaviour is runtime implementation dependent.
- ***requires : requires (0..n)*** - A service element has ***zero or more requires subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.
- ***policySetAttachment : policySetAttachment (0..n)*** - A service element has ***zero or more policySetAttachment subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.

5.1.1 Service Examples

Figure 5-1 shows the service symbol that used to represent a service in an assembly diagram:

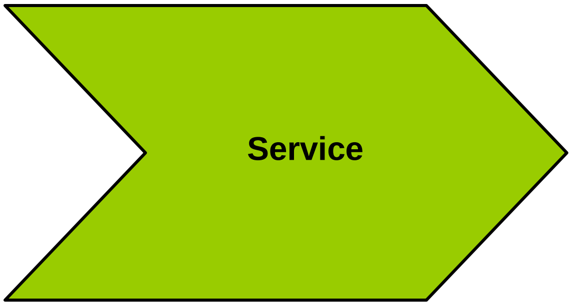


Figure 5-1: Service symbol

Figure 5-2 shows the assembly diagram for the MyValueComposite containing the service MyValueService.

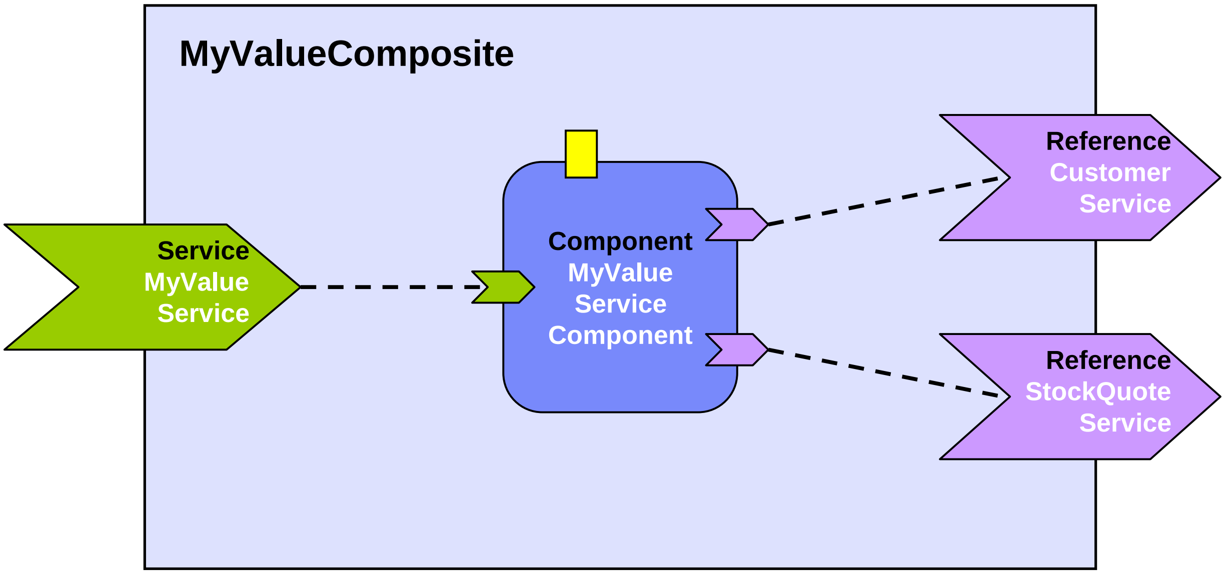


Figure 5-2: MyValueComposite showing Service

Snippet 5-3 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/200912"
                targetNamespace="http://foo.com"
                name="MyValueComposite" >

   ...

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

Snippet 5-3: Example composite with a service

5.2 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. Snippet 5-4 shows the composite pseudo-schema with the pseudo-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/200912" … >
   …
   <reference name="xs:NCName" target="list of xs:anyURI"?
       promote="list of xs:anyURI" wiredByImpl="xs:boolean"?
       multiplicity="0..1 or 1..1 or 0..n or 1..n"
       requires="list of xs:QName"? policySets="list of xs:QName"?>*
       <interface … />?
       <binding … />*
       <callback>?
          <binding … />+
       </callback>
       <requires/>*
       <policySetAttachment/>*
   </reference>
   …
</composite>
```

Snippet 5-4: composite Pseudo-Schema with reference Child Element

The ***composite reference*** element has the ***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 than 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 reference name can be omitted 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] <include/> processing MUST take place before the processing of the @promote attribute of a composite reference is performed. [ASM60037]

  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 each of the component reference interfaces MUST be a compatible 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 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 : listOfQNames (0..1)*** – a list of policy intents. See the Policy Framework specification [SCA-POLICY] for a description of this attribute. Specified intents add to or further qualify the intents defined for the promoted component reference.
- ***policySets : listOfQNames (0..1)*** – a list of policy sets. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.
- ***multiplicity : (1..1)*** - Defines the number of wires that can connect the reference to target services. The multiplicity of a composite reference is always specified explicitly and can have one of 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 multiplicity of a composite reference MUST be equal to or further restrict the multiplicity of each of the component references that it promotes, with the exception that the multiplicity of the composite reference does not have to require a target if there is already a target on the component reference. This means that a component reference with multiplicity 1..1 and a target can be promoted by a composite reference with multiplicity 0..1, and a component reference with multiplicity 1..n and one or more targets can be promoted by a composite reference with multiplicity 0..n or 0..1. [ASM60011]

  The valid values for composite reference multiplicity are shown in the following tables:

Composite Reference multiplicity	Component Reference multiplicity (where there are no targets declared)			
	0..1	1..1	0..n	1..n
0..1	YES	NO	YES	NO
1..1	YES	YES	YES	YES
0..n	NO	NO	YES	NO
1..n	NO	NO	YES	YES

Composite Reference multiplicity	Component Reference multiplicity (where there are targets declared)			
	0..1	1..1	0..n	1..n
0..1	YES	YES	YES	YES
1..1	YES	YES	YES	YES
0..n	NO	NO	YES	YES
1..n	NO	NO	YES	YES

- ***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. If set to "true" it indicates that the target of the reference is set at runtime by the implementation code (for example 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 is not intended to be wired statically within a using composite, but left unwired. All the component references promoted by a single composite reference MUST have the same value for @wiredByImpl. [ASM60035] If the @wiredByImpl attribute is not specified on the composite reference, the default value is "true" if all of the promoted component references have a wiredByImpl value of "true", and the default value is "false" if all the promoted component references have a wiredByImpl value of "false". If the @wiredByImpl attribute is specified, its value MUST be "true" if all of the promoted component references have a wiredByImpl value of "true", and its value MUST be "false" if all the promoted component references have a wiredByImpl value of "false". [ASM60036]

The ***composite reference*** element has the ***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). [ASM60012] If no interface is declared on a composite reference, the interface from one of its promoted component references MUST be used for the component type associated with the composite. [ASM60013] For details on the interface element see the Interface section.
- ***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 "Specifying the Target Service(s) for a Reference".
- ***callback (0..1) / binding : Binding (1..n)*** - A ***callback*** element is used if the interface has a callback defined and the callback element has one or more ***binding*** elements as subelements. The ***callback*** and its binding subelements are specified if there is a need to have binding details used to handle callbacks. Callback binding elements attached to the composite reference override any callback binding elements defined on any of the promoted component references. If the callback element is not present on the composite service, any callback binding elements that are declared on all the promoted references are used. If the callback element is not present at all, the behaviour is runtime implementation dependent.
- ***requires : requires (0..n)*** - A service element has ***zero or more requires subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.
- ***policySetAttachment : policySetAttachment (0..n)*** - A service element has ***zero or more policySetAttachment subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.

5.2.1 Example Reference

Figure 5-3 shows the reference symbol that is used to represent a reference in an assembly diagram.

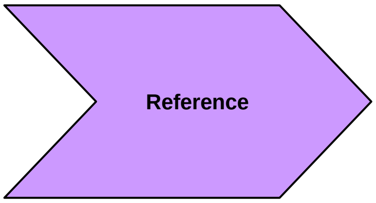


Figure 5-3: Reference symbol

Figure 5-4 shows the assembly diagram for the MyValueComposite containing the reference CustomerService and the reference StockQuoteService.

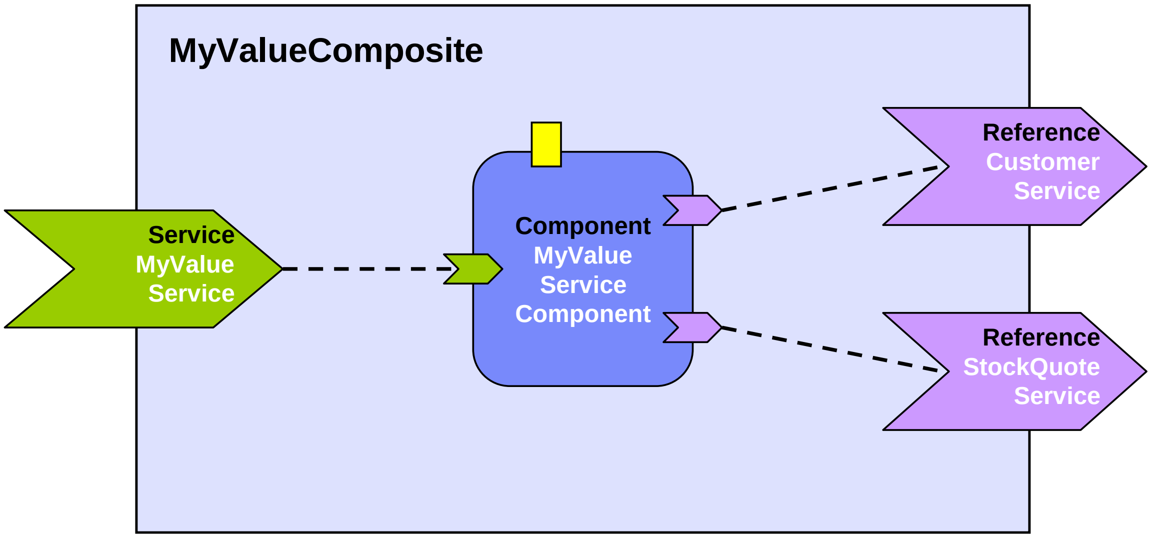


Figure 5-4: MyValueComposite showing References

Snippet 5-5 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/200912"
                targetNamespace="http://foo.com"
                name="MyValueComposite" >

  ...

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

```

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

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

   ...

</composite>
```

Snippet 5-5: Example composite with a reference

5.3 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 is 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.

Snippet 5-6 shows the composite pseudo-schema with the pseudo-schema for a ***reference*** element:

```
<?xml version="1.0" encoding="ASCII"?>
<!-- Composite Property schema snippet -->
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912" … >
   …
   <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
         many="xs:boolean"? mustSupply="xs:boolean"?>*
         default-property-value?
   </property>
   …
</composite>
```

Snippet 5-6: composite Pseudo-Schema with property Child Element

The ***composite property*** element has the ***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

    A single property element MUST NOT contain both a @type attribute and an @element attribute. [ASM60040]
- ***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.

The property element can contain a ***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 (e.g. 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.

5.3.1 Property Examples

For the example Property declaration and value setting in Snippet 5-8, the complex type in Snippet 5-7 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="tns:MyComplexType"/>
   <xsd:complexType name="MyComplexType">
         <xsd:sequence>
                <xsd:element name="a" type="xsd:string"/>
                <xsd:element name="b" type="xsd:anyURI"/>
         </xsd:sequence>
         <attribute name="attr" type="xsd:string" use="optional"/>
   </xsd:complexType>
</xsd:schema>
```

Snippet 5-7: Complex Type for Snippet 5-8

The composite in Snippet 5-8 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/200912"
                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>
```

Snippet 5-8: Example property with a Complext Type

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 of type foo:MyComplexType and it has 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.

Snippet 5-9 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/200912"
                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>
```

Snippet 5-9: Example property with a Simple Type

In the example in Snippet 5-9, 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:

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

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

Snippet 5-10: Example property with a Simple Type and Default Value

– 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>
```

Snippet 5-11: Example property with a Complex Type and Default Value

– 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>
```

Snippet 5-12: Example property with a Global Element Type

5.4 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.

Snippet 5-13 shows the composite pseudo-schema with the pseudo-schema for the wire child element:

```
<!-- Wires schema snippet -->
<composite ...>
   ...
   <wire source="xs:anyURI" target="xs:anyURI" replace="xs:boolean"?/>*
   ...
</composite>
```

Snippet 5-13: composite Pseudo-Schema with wire Child Element

The ***reference element of a component*** 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> [/<binding-name>]?]?
    - <component-name> is the name of the target component.
    - <service-name> is the name of the target service within the component.

      If <service-name> is present, the component service with @name corresponding to <service-name> MUST be used for the wire. [ASM60046]

      If there is no component service with @name corresponding to <service-name>, the SCA runtime MUST raise an error. [ASM60047]

      If <service-name> is not present, the target component MUST have one and only one service with an interface that is a compatible superset of the wire source's interface and satisifies the policy requirements of the wire source, and the SCA runtime MUST use this service for the wire. [ASM60048]
    - <binding-name> is the name of the service's binding to use. The <binding-name> can be the default name of a binding element (see section 8 "Binding").

      If <binding-name> is present, the <binding/> subelement of the target service with @name corresponding to <binding-name> MUST be used for the wire. [ASM60049] If there is no <binding/> subelement of the target service with @name corresponding to <binding-name>, the SCA runtime MUST raise an error. [ASM60050] If <binding-name> is not present and the target service has multiple <binding/> subelements, the SCA runtime MUST choose one and only one of the <binding/> elements which satisfies the mutual policy requirements of the reference and the service, and the SCA runtime MUST use this binding for the wire. [ASM60051]

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

- ***source (1..1)*** – names the source component reference. The valid URI scheme is:
    - <component-name>[/<reference-name>]?
        - where the source is a component reference. The reference name can be omitted if the source component only has one reference
- ***target (1..1)*** – names the target component service. The valid URI scheme is the same as the one defined for component references above.
- ***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, where the reference is on a component at the Domain level.

<include/> processing MUST take place before the @source and @target attributes of a wire are resolved. [ASM60039]

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.

The interface declared by the target of a wire MUST be a compatible superset of the interface declared by the source of the wire. [ASM60043] See the section on Interface Compatibility for a definition of "compatible superset".

A Wire can connect between different interface languages (e.g. 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 might only have the business interface of the reference and might 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 is encouraged to issue a warning.

5.4.1 Wire Examples

Figure 5-5 shows the assembly diagram for the MyValueComposite2 containing wires between service, components and references.

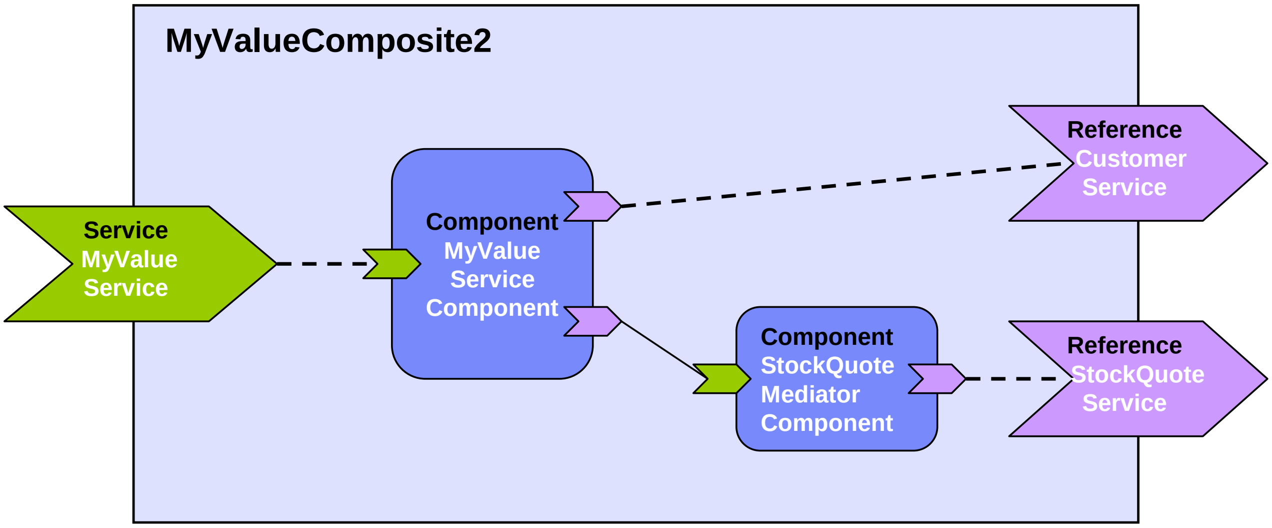

MyValueComposite2
Service MyValue Service
Component MyValue Service Component
Component StockQuote Mediator Component
Reference Customer Service
Reference StockQuote Service

Figure 5-5: MyValueComposite2 showing Wires

Snippet 5-14 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/200912"
                targetNamespace="http://foo.com"
                name="MyValueComposite2" >
```

```

   <service name="MyValueService" promote="MyValueServiceComponent">
         <interface.java interface="services.myvalue.MyValueService"/>
         <binding.ws wsdlElement="http://www.myvalue.org/MyValueService#
                wsdl.port(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 wsdlElement="http://www.stockquote.org/StockQuoteService#
                wsdl.port(StockQuoteService/StockQuoteServiceSOAP)"/>
   </reference>

</composite>
```

Snippet 5-14: Example composite with a wire

5.4.2 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 SCA runtime MUST search within the composite for target services which have an interface that is a compatible superset of the interface of the reference. [ASM60022]

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 [SCA-POLICY] 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]

5.4.3 Autowire Examples

Snippet 5-15 and Snippet 5-16 demonstrate 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.

Figure 5-6 is a diagram for the composite:

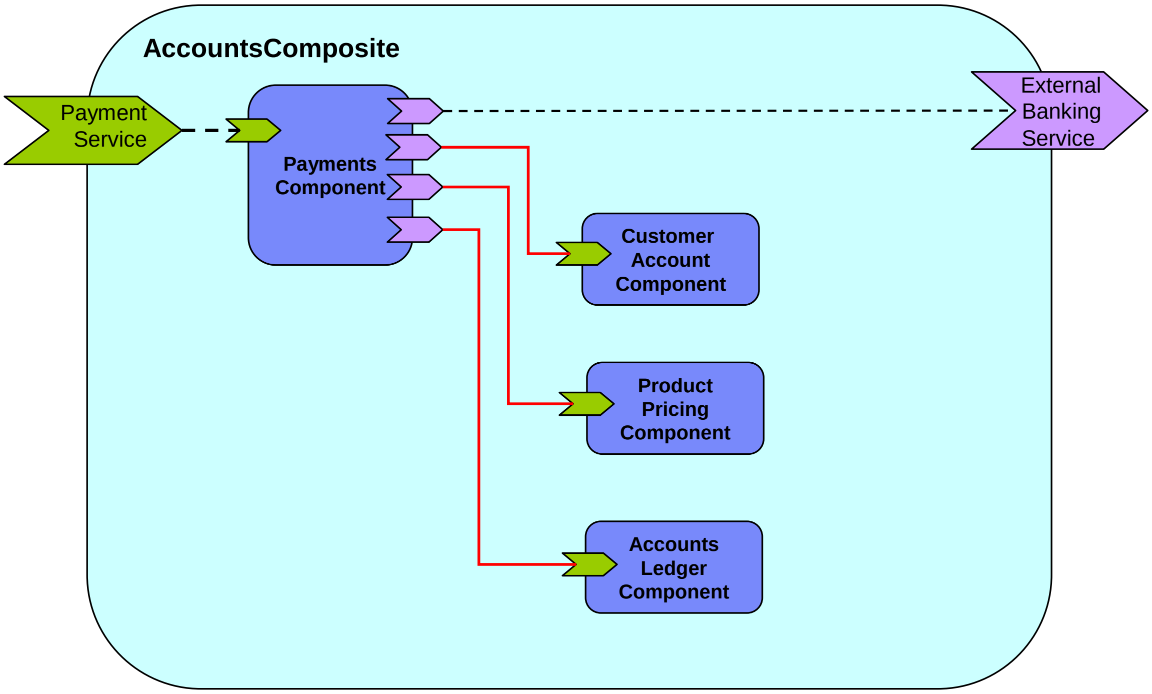

AccountsComposite
Payment Service
Payments Component
External Banking Service
Customer Account Component
Product Pricing Component
Accounts Ledger Component

Figure 5-6: Example Composite for Autowire

Snippet 5-15 is 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/200912"
    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>
```

Snippet 5-15: Example composite with Explicit wires

Snippet 5-16 is 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/200912"
     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>
```

Snippet 5-16: composite of Snippet 5-15 Using autowire

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.

5.5 Consumer

The ***consumers of a composite*** are defined by ***promoting*** consumers defined by components contained in the composite. Consumers are promoted by means of a composite ***consumer element***, which is a child element of the composite element. Promotion of the component consumer allows the configuration of the composite consumer set by a higher level component to override the configuration of the lower component consumer. There can be ***zero or more*** consumer elements in a composite.

Every event received by the composite consumer is sent on to all of the promoted consumers.

Snippet 5-17 shows the pseudo-schema for a composite consumer element:

```
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912" … >
   …

   <consumer name="xs:NCName"
         promote="list of xs:anyURI"
         requires="list of xs:QName"?
         policySets="list of xs:QName"?>*
         <filters/>?
         <requires/>*
         <policySetAttachment/>*
   </consumer>
   …
</composite>
```

Snippet 5-17: composite Pseudo-Schema with consumer Child Element

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

- ***name: NCName (1..1)*** – the name of the consumer. The name of the consumer MUST be unique amongst the consumer elements of the composite. [ASM60101] The name the composite consumer can be different from the name of the promoted component consumer.
- ***promote: listOfAnyURI (1..1)*** – identifies the promoted consumers. The value is a list containing entries of the form componentName/consumerName. The consumer name is optional if the component only has one consumer. The same component consumer can be promoted by more than one composite consumer. A composite <consumer/> element's @promote attribute MUST identify one of the component consumers within that composite. [ASM60102] <include/> processing MUST take place before the processing of the @promote attribute of a composite service is performed. [ASM60103]
- ***requires : listOfQNames (0..1)*** – a list of policy intents. See the Policy Framework specification [SCA-POLICY] for a description of this attribute. Specified intents add to or further qualify the required intents defined by the promoted component consumer.
- ***policySets : listOfQNames (0..1)*** – a list of policy sets. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.

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

- ***filters: Filters (0..1) –*** filter elements. See the section Filters: Selecting Subsets of Events for a detailed description of filters.
- ***requires : requires (0..n)*** - A service element has ***zero or more requires subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.
- ***policySetAttachment : policySetAttachment (0..n)*** - A producer element has ***zero or more policySetAttachment subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.

5.6 Producer

The ***producers of a composite*** are defined by ***promoting*** producers defined by components contained in the composite. Producers are promoted by means of a composite ***producer element***, which is a child element of the composite element. Promotion of the component producer allows the configuration of the composite producer set by a higher level component to override the configuration of the lower component producer. There can be ***zero or more*** producer elements in a composite.

Every event sent by any of the composite producer is sent out by the promoted producer.

Snippet 5-18 shows the pseudo-schema for a composite producer element:

```
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912" … >
   …

   <producer name="xs:NCName"
         promote="list of xs:anyURI"
         requires="list of xs:QName"?
         policySets="list of xs:QName"?>*
         <eventType/>?
         <requires/>*
         <policySetAttachment/>*
   </producer>

   …
</composite>
```

Snippet 5-18: composite Pseudo-Schema with producer Child Element

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

- ***name: NCName (1..1)*** – the name of the producer. The name of the producer MUST be unique amongst the producer elements of the composite. [ASM60104] The name the composite producer can be different from the name of the promoted component producer.
- ***promote: listOfAnyURI (1..1)*** – identifies the promoted producers. The value is a list containing entries of the form componentName/producerName. The prodcuer name is optional if the component only has one producer. The same component producer can be promoted by more than one composite producer. A composite <producer/> element's @promote attribute MUST identify one of the component producers within that composite. [ASM60105] <include/> processing MUST take place before the processing of the @promote attribute of a composite service is performed. [ASM60106]
- ***requires : listOfQNames (0..1)*** – a list of policy intents. See the Policy Framework specification [SCA-POLICY] for a description of this attribute. Specified intents add to or further qualify the required intents defined by the promoted component producer.
- ***policySets : listOfQNames (0..1)*** – a list of policy sets. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.
- ***promotes (required)*** – identifies the promoted producers. The value is a list containing entries of the form componentName/producerName. The producer name is optional if the component only has one producer.

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

- ***eventType : EventType (0..1)*** - A producer has ***zero or one eventType*** child subelement. See Section Use of <eventType> on a Producer.
- ***requires : requires (0..n)*** - A service element has ***zero or more requires subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.

- ***policySetAttachment : policySetAttachment (0..n)*** - A producer element has ***zero or more policySetAttachment subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.

5.7 Channels

A ***channel*** is an SCA artifact that is used to connect a set of event producers to a set of event consumers. The channel can accept events sent by many producers and it can send all of these events to each of the set of consumers, which are subscribed to the channel.

One role of the channel is to act as an intermediary between the set of producers and the set of consumers. The channel exists separately from any individual producer or consumer.

A channel acts as if it has a single consumer element with the name "in", to which producers can send events. A channel acts as if it has a single producer element with the name "out", from which subscribers receive events.

A channel may be configured with filters, which defines the set of events that the channel accepts. If an event does not match the filters defined, the event is discarded. See section Filters: Selecting Subsets of Events for more details.

The pseudo-schema for Channels is shown in Snippet 5-19.

```
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912" … >
  …

  <channel name="xs:NCName"
         requires="list of xs:QName"?
         policySets="list of xs:QName"?>*
         <filters/>?
         <binding … />?
         <requires/>*
         <policySetAttachment/>*
  </channel>

  …
</composite>
```

Snippet 5-19: composite Pseudo-Schema with channel Child Element

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

- ***name: NCName (1..1)*** – the name of the channel. The name of the channel MUST be unique amongst the channel elements of the composite. [ASM60107]
- ***requires : listOfQNames (0..1)*** – a list of policy intents that apply to the handling of events by this channel. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.
- ***policySets : listOfQNames (0..1)*** – a list of policy sets that apply to the handling of events by this channel. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.

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

- ***filters: Filters (0..1) –*** filter elements. See the section Filters: Selecting Subsets of Events for a detailed description of filters.
- ***binding : Binding (0..1)*** - A channel element has zero or one ***binding element*** as children. Each element defines the mechanism used for transmission of events from/to this channel. Details of the binding element are described in the Bindings section.
- ***requires : requires (0..n)*** - A service element has ***zero or more requires subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.

- ***policySetAttachment : policySetAttachment (0..n)*** - A channel element has ***zero or more policySetAttachment subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.

5.8 Scopes of Channels

Channels can exist either at the Domain level or they can exist within a composite used as an implementation.

Channels at the Domain level (i.e., channels that are present in the domain-level composite) are termed ***domain channels***. They can be used as targets for producers at any level within the composition hierarchy. They can be used as sources for consumers at any level within the composition hierarchy. An SCA runtimes MUST support the use of domain channels. [ASM60108]. To create a Domain Channel, deploy a composite containing a channel directly to the SCA Domain (i.e., do not use that composite as the implementation of some component in the Domain).

Channels within a composite used as an implementation are private to the components within that composite. These ***private channels*** can only be the targets for producers existing within the same composite as the channel. Private channels can only be sources for consumers existing withing the same composite as the channel. An SCA runtime MAY support the use of private channels. [ASM60109].

This division of Channels into global channels and private channels permits the assembler of an application to control the set of components involved in event exchange, if required. Producers and consumers of global channels are effectively uncontrolled – they exist at the Domain and they can be added or removed at any time through deployment actions. Private channels have restricted sets of producers and consumers – these sets are decided by the assembler when the composite containing them is created.

5.9 The Default Domain Channel

In SCA Event processing, there is a special ***default channel*** which is a domain channel.

The default channel always exists, even if it not declared explicitly in the configuration of the Domain. The default channel has the URI "/".

Producers and consumers at any level in the Domain can communicate using the default channel by using the URI "/" in their target or source attribute respectively.

5.10 The URI of a Channel

When used for the source of a consumer or for the target of a producer, a channel is referenced by a URI. The URI of a channel is built from the name of the channel.

The URI of a private channel is the name of the channel. For example, "local-weather"

The URI of a domain channel is "/" followed by the name of the channel. For example, "/cyclones"

The URI of the default domain channel is simply "/".

5.11 Using Composites as Component Implementations

Composites can be used as ***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, connect consumers and producers of the composite to channels, 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 also needs to 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. [ASM60033] (according to the various rules for specifying target services for a component reference described in the section " Specifying the Target Service(s) for a Reference").
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, composite consumer elements, composite producer 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. Snippet 5-20 shows the pseudo-schema for the implementation.composite element:

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

Snippet 5-20: implementation.composite Pseudo-Schema

The ***implementation.composite*** element has the 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 : listOfQNames (0..1)*** – a list of policy intents. See the Policy Framework specification [SCA-POLICY] for a description of this attribute. Specified intents add to or further qualify the required intents defined for the promoted component reference.
- ***policySets : listOfQNames (0..1)*** – a list of policy sets. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.

5.11.1 Component Type of a Composite used as a Component Implementation

An SCA runtime MUST introspect the componentType of a Composite used as a Component Implementation following the rules defined in the section "Component Type of a Composite used as a Component Implementation" [ASM60045]

The componentType of a Composite used as a Component Implementation is introspected from the Composite document as follows:

A <service/> element exists for each direct <service/> subelement of the <composite/> element

- @name attribute set to the value of the @name attribute of the <service/> in the composite

- @requires attribute set to the value of the @requires attribute of the <service/> in the composite, if present (the value of the @requires attribute contains the intents which apply to the promoted component service, as defined in the Policy Framework specification [SCA_POLICY]). If no intents apply to the <service/> in the composite, the @requires attribute is omitted.
- @policySets attribute set to the value of the @policySets attribute of the <service/> in the composite, if it is present. If the @policySets attribute of the <service/> element in the composite is absent, the @policySets attribute is omitted.
- <interface/> subelement set to the <interface/> subelement of the <service/> element in the composite. If not declared on the composite service, it is set to the <interface/> subelement which applies to the component service which is promoted by the composite service (this is either an explicit <interface/> subelement of the component <service/>, or the <interface/> element of the corresponding <service/> in the componentType of the implementation used by the component).
- <binding/> subelements set to the <binding/> subelements of the <service/> element in the composite. If not declared on the composite service, the <binding/> subelements which apply to the component service promoted by the composite service are used, if any are present. If none are present in both of these locations, <binding/> subelements are omitted.
- <callback/> subelement is set to the <callback/> subelement of the <service/> element in the composite. If no <callback/> subelement is present on the composite <service/> element, the <callback/> subelement is omitted.

A <reference/> element exists for each direct <reference/> subelement of the <composite/> element.

- @name attribute set to the value of the @name attribute of the <reference/> in the composite
- @requires attribute set to the value of the @requires attribute of the <reference/> in the composite, if present (the value of the @requires attribute contains the intents which apply to the promoted component references, as defined in the Policy Framework specification [SCA_POLICY]). If no intents apply to the <reference/> in the composite, the @requires attribute is omitted.
- @policySets attribute set to the value of the @policySets attribute of the <reference/> in the composite, if present. If the @policySets attribute of the <reference/> element in the composite is absent, the @policySets attribute is omitted.
- @target attribute is set to the value of the @target attribute of the <reference/> in the composite, if present, otherwise the @target attribute is omitted.
- @wiredByImpl attribute is set to the value of the @wiredByImpl attribute of the <reference/> in the composite, if present. If it is not declared on the composite reference, it is set to the value of the @wiredByImpl attribute of the promoted reference(s).
- @multiplicity attribute is set to the value of the @multiplicity attribute of the <reference/> in the composite
- <interface/> subelement set to the <interface/> subelement of the <reference/> element in the composite. If not declared on the composite reference, it is set to the <interface/> subelement which applies to one of the component reference(s) which are promoted by the composite reference (this is either an explicit <interface/> subelement of the component <reference/>, or the <interface/> element of the corresponding <reference/> in the componentType of the implementation used by the component).
- <binding/> subelements set to the <binding/> subelements of the <reference/> element in the composite. Otherwise, <binding/> subelements are omitted.
- <callback/> subelement is set to the <callback/> subelement of the <reference/> element in the composite. Otherwise, <callback/> subelements are omitted.

A <consumer/> element exists for each direct <consumer/> subelement of the <composite/> element

- @name attribute set to the value of the @name attribute of the <consumer/> in the composite
- @requires attribute set to the value of the @requires attribute of the <consumer/> in the composite, if present (the value of the @requires attribute contains the intents which apply to the

promoted component consumer, as defined in the Policy Framework specification [SCA_POLICY]). If no intents apply to the <consumer/> in the composite, the @requires attribute is omitted.

- @policySets attribute set to the value of the @policySets attribute of the <consumer/> in the composite, if it is present. If the @policySets attribute of the <consumer/> element in the composite is absent, the @policySets attribute is omitted.
- <filters/> subelement set to the <filters/> subelement of the <consumer/> element in the composite, if present (the value of the <filters> element contains the filters that apply to the promoted component consumer, as defined by the Section Filters: Selecting Subsets of Events). If no filters apply to the <consumer/> in the composite, the <filters> subelement is omitted.
- <binding/> subelements set to the <binding/> subelements of the <consumer/> element in the composite. If not declared on the composite consumer, the <binding/> subelements which apply to the component consumer promoted by the composite consumer are used, if any are present. If none are present in both of these locations, <binding/> subelements are omitted.

A <producer/> element exists for each direct <producer/> subelement of the <composite/> element

- @name attribute set to the value of the @name attribute of the <producer/> in the composite
- @requires attribute set to the value of the @requires attribute of the <producer/> in the composite, if present (the value of the @requires attribute contains the intents which apply to the promoted component producer, as defined in the Policy Framework specification [SCA_POLICY]). If no intents apply to the <producer/> in the composite, the @requires attribute is omitted.
- @policySets attribute set to the value of the @policySets attribute of the <producer/> in the composite, if it is present. If the @policySets attribute of the <producer/> element in the composite is absent, the @policySets attribute is omitted.
- <eventType/> subelement set to the <eventType/> subelement of the <prodcuer/> element in the composite, if present (the value of the <eventType> element contains the event types that apply to the promoted component producer, as defined by the Section Representation of Events and Event Types. If no event types apply to the <producer/> in the composite, the <eventType> subelement is omitted.
- <binding/> subelements set to the <binding/> subelements of the <producer/> element in the composite. If not declared on the composite producer, the <binding/> subelements which apply to the component producer promoted by the composite producer are used, if any are present. If none are present in both of these locations, <binding/> subelements are omitted.

A <property/> element exists for each direct <property/> subelement of the <composite/> element.

- @name attribute set to the value of the @name attribute of the <property/> in the composite
- @type attribute set to the value of the @type attribute of the <property/> in the composite, if present
- @element attribute set to the value of the @element attribute of the <property/> in the composite, if present
  (Note: either a @type attribute is present or an @element attribute is present - one of them has to be present, but both are not allowed)
- @many attribute set to the value of the @many attribute of the <property/> in the composite, if present, otherwise omitted.
- @mustSupply attribute set to the value of the @mustSupply attribute of the <property/> in the composite, if present, otherwise omitted.
- @requires attribute set to the value of the @requires attribute of the <property/> in the composite, if present, otherwise omitted.
- @policySets attribute set to the value of the @policySets attribute of the <property/> in the composite, if present, otherwise omitted.

A <implementation/> element exists if the <composite/> element has either of the @requires or @policySets attributes declared, with:

- @requires attribute set to the value of the @requires attribute of the composite, if present, otherwise omitted.
- @policySets attribute set to the value of he @policySets attribute of the composite, if present, otherwise omitted.

5.11.2 Example of Composite used as a Component Implementation

Snippet 5-21 shows 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/200912
    file:/C:/Strategy/SCA/v09_osoaschemas/schemas/sca.xsd"
    xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
    targetNamespace="http://foo.com"
    xmlns:foo="http://foo.com"
    name="AccountComposite">

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

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

    <consumer name="StockQuoteListener" promote="AccountServiceComponent" />

    <producer name="AccountServiceEvents" promote="AccountServiceComponent" />

    <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>
```

Snippet 5-21: Example of a composite Using implementation.composite

5.12 Using Composites through Inclusion

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

A composite can 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** 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, follow the rules defined for that extension. Authors of attribute extensions on the composite element define the rules applying to those attributes 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 an error. [ASM60041]

The composite file used for inclusion can have any contents. The composite element can 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 can 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 (e.g. 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.

Snippet 5-22 snippet shows the pseudo-schema for the include element:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Include snippet -->
<composite ...>
   ...
   <include name="xs:QName"/>*
   ...
</composite>
```

Snippet 5-22: include Pseudo-Schema

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

- ***name: QName (1..1)*** – the name of the composite that is included. The @name attribute of an include element MUST be the QName of a composite in the SCA Domain. [ASM60042]

5.12.1 Included Composite Examples

Figure 5-7 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. (TODO: include new example with pub-sub)

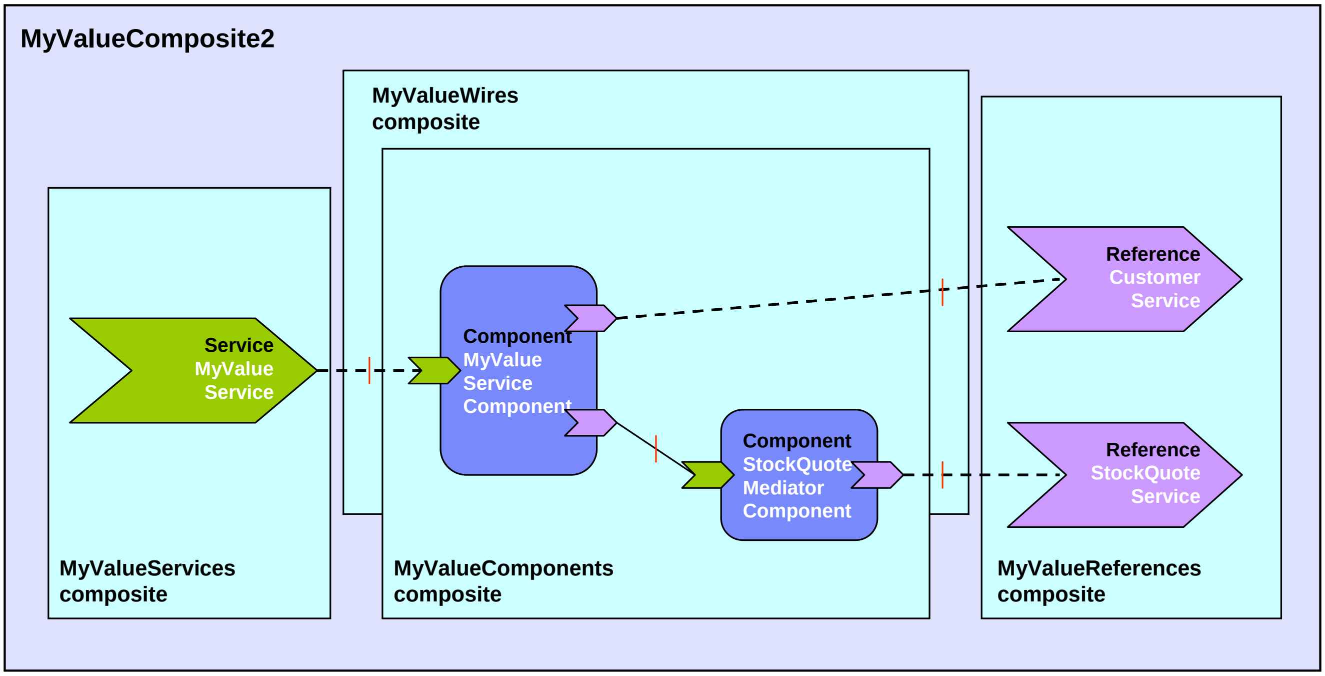

MyValueComposite2
MyValueWires composite
Service MyValue Service
Component MyValue Service Component
Reference Customer Service
Component StockQuote Mediator Component
Reference StockQuote Service
MyValueServices composite
MyValueComponents composite
MyValueReferences composite

Figure 5-7 MyValueComposite2 built from 4 included composites

Snippet 5-23 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/200912"
                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>
```

Snippet 5-23: Example composite with includes

Snippet 5-24 shows the content of the MyValueServices.composite file.

```
<?xml version="1.0" encoding="ASCII"?>
<composite      xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
                targetNamespace="http://foo.com"
                xmlns:foo="http://foo.com"
                name="MyValueServices" >

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

</composite>
```

Snippet 5-24: Example Partial composite with Only a service

Snippet 5-25 shows the content of the MyValueComponents.composite file.

```
<?xml version="1.0" encoding="ASCII"?>
<composite      xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
                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>
```

Snippet 5-25: Example Partial composite with Only components

Snippet 5-26 shows the content of the MyValueReferences.composite file.

```
<?xml version="1.0" encoding="ASCII"?>
<composite      xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
                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 wsdlElement="http://www.stockquote.org/StockQuoteService#
             wsdl.port(StockQuoteService/StockQuoteServiceSOAP)"/>
   </reference>

</composite>
```

Snippet 5-26: Example Partial composite with Only references

Snippet 5-27 shows the content of the MyValueWires.composite file.

```
<?xml version="1.0" encoding="ASCII"?>
<composite      xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
                targetNamespace="http://foo.com"
                xmlns:foo="http://foo.com"
                name="MyValueWires" >

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

</composite>
```

Snippet 5-27: Example Partial composite with Only a wire

5.13 Composites which Contain Component Implementations of Multiple Types

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

5.14 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 can 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 is used to separately identify each use of a component - for example, the structural URI can 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)
- Consumer:
  #consumer(consumername)
- Producer:
  #producer(producername)
- Consumer binding:
  #consumer-binding(consumername/bindingname)
- Producer binding:
  #producer-binding(producername/bindingname)

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

6 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 [WSDL-11])
- 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.

Snippet 6-1 shows the pseudo-schema for the ***interface*** base element:

```
<interface remotable="boolean"? requires="list of xs:QName"?
           policySets="list of xs:QName"?>
   <requires/>*
   <policySetAttachment/>*
</interface>
```

Snippet 6-1: interface Pseudo-Schema

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

- ***remotable : boolean (0..1)*** – indicates whether an interface is remotable or not (see the section on Local and Remotable interfaces). A value of “true” means the interface is remotable, and a value of “false” means it is not. The @remotable attribute has no default value. This attribute is used as an alternative to interface type specific mechanisms such as the @Remotable annotation on a Java interface. The remotable nature of an interface in the absence of this attribute is interface type specific. The rules governing how this attribute relates to interface type specific mechanisms are defined by each interface type. When specified on an interface definition which includes a callback, this attribute also applies to the callback interface (see the section on Bidirectional Interfaces).
- ***requires : listOfQNames (0..1)*** – a list of policy intents. See the Policy Framework specification [SCA-POLICY] for a description of this attribute
- ***policySets : listOfQNames (0..1)*** – a list of policy sets. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.

The ***interface*** element has the following ***subelements***:

- ***requires : requires (0..n)*** - A service element has ***zero or more requires subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.
- ***policySetAttachment : policySetAttachment (0..n)*** - A service element has ***zero or more policySetAttachment subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.

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].

6.1 Local and Remotable Interfaces

A remotable service is one which can 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 [WSI-BP] (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]

Snippet 6-2 shows an example of a remotable java interface:

```
package services.hello;

@Remotable
public interface HelloService {

   String hello(String message);
}
```

Snippet 6-2: Example remotable interface

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***.

6.2 Interface Compatibility

The ***compatibility*** of two interfaces is defined in this section and these definitions are used throughout this specification. Three forms of compatibility are defined:

- Compatible interfaces
- Compatible subset
- Compatible superset

Note that WSDL 1.1 message parts can point to an XML Schema element declaration or to an XML Schema types. When determining compatibility between two WSDL operations, a message part that points to an XML Schema element declaration is considered to be incompatible with a message part that points to an XML Schema type.

6.2.1 Compatible Interfaces

An interface A is ***Compatible*** with a second interface B if and only if all of points 1 through 7 in the following list apply:

1. interfaces A and B are either both remotable or else both local
2. the set of operations in interface A is the same as the set of operations in interface B
3. compatibility for individual operations of the interfaces A and B is defined as compatibility of the signature, i.e., the operation name, the input types, and the output types are the same
4. the order of the input and output types for each operation in interface A is the same as the order of the input and output types for the corresponding operation in interface B
5. the set of Faults and Exceptions expected by each operation in interface A is the same as the set of Faults and Exceptions specified by the corresponding operation in interface B
6. for checking the compatibility of 2 remotable interfaces which are in different interface languages, both are mapped to WSDL 1.1 (if not already WSDL 1.1) and compatibility checking is done between the WSDL 1.1 mapped interfaces.

   For checking the compatibility of 2 local interfaces which are in different interface languages, the method of checking compatibility is defined by the specifications which define those interface types, which must define mapping rules for the 2 interface types concerned.
7. if either interface A or interface B declares a callback interface then both interface A and interface B declare callback interfaces and the callback interface declared on interface A is compatible with the callback interface declared on interface B, according to points 1 through 6 above

6.2.2 Compatible Subset

An interface A is a ***Compatible Subset*** of a second interface B if and only if all of points 1 through 7 in the following list apply:

1. interfaces A and B are either both remotable or else both local
2. the set of operations in interface A is the same as or is a subset of the set of operations in interface B
3. compatibility for individual operations of the interfaces A and B is defined as compatibility of the signature, i.e., the operation name, the input types, and the output types are the same

4. the order of the input and output types for each operation in interface A is the same as the order of the input and output types for the corresponding operation in interface B
5. the set of Faults and Exceptions expected by each operation in interface A is the same as or is a superset of the set of Faults and Exceptions specified by the corresponding operation in interface B
6. for checking the compatibility of 2 remotable interfaces which are in different interface languages, both are mapped to WSDL 1.1 (if not already WSDL 1.1) and compatibility checking is done between the WSDL 1.1 mapped interfaces.

   For checking the compatibility of 2 local interfaces which are in different interface languages, the method of checking compatibility is defined by the specifications which define those interface types, which must define mapping rules for the 2 interface types concerned.
7. if either interface A or interface B declares a callback interface then both interface A and interface B declare callback interfaces and the callback interface declared on interface B is a compatible subset of the callback interface declared on interface A, according to points 1 through 6 above

6.2.3 Compatible Superset

An interface A is a ***Compatible Superset*** of a second interface B if and only if all of points 1 through 7 in the following list apply:

1. interfaces A and B are either both remotable or else both local
2. the set of operations in interface A is the same as or is a superset of the set of operations in interface B
3. compatibility for individual operations of the interfaces A and B is defined as compatibility of the signature, i.e., the operation name, the input types, and the output types are the same
4. the order of the input and output types for each operation in interface B is the same as the order of the input and output types for the corresponding operation in interface A
5. the set of Faults and Exceptions expected by each operation in interface A is the same as or is a subset of the set of Faults and Exceptions specified by the corresponding operation in interface B
6. for checking the compatibility of 2 remotable interfaces which are in different interface languages, both are mapped to WSDL 1.1 (if not already WSDL 1.1) and compatibility checking is done between the WSDL 1.1 mapped interfaces.

   For checking the compatibility of 2 local interfaces which are in different interface languages, the method of checking compatibility is defined by the specifications which define those interface types, which must define mapping rules for the 2 interface types concerned.
7. if either interface A or interface B declares a callback interface then both interface A and interface B declare callback interfaces and the callback interface declared on interface B is a compatible superset of the callback interface declared on interface A, according to points 1 through 6 above

6.3 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 a callback interface. If a callback interface is specified, SCA refers to the interface as a whole as a bidirectional interface.

Snippet 6-3 shows the interface element defined using Java interfaces with a @callbackInterface attribute.

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

Snippet 6-3: Example interface with a callback

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]

See the section on Interface Compatibility 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.

6.4 Long-running Request-Response Operations

6.4.1 Background

A service offering one or more operations which map to a WSDL request-response pattern might 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 could 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 performs 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.

6.4.2 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.

6.4.3 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.

6.4.4 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 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.

6.4.5 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.

6.5 SCA-Specific Aspects for WSDL Interfaces

There are a number of aspects that SCA applies to interfaces in general, such as marking them as having a callback interface. 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/200912) needs to be declared within the WSDL document.

First, SCA defines a global element in the SCA namespace which provides a mechanism to attach policy intents - ***requires***. Snippet 6-4 shows the definition of the requires element:

```
   <element name="requires">
      <complexType>
         <sequence minOccurs="0" maxOccurs="unbounded">
            <any namespace="##other" processContents="lax"/>
         </sequence>
         <attribute name="intents" type="sca:listOfQNames" use="required"/>
         <anyAttribute namespace="##other" processContents="lax"/>
      </complexType>
   </element>

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

Snippet 6-4: requires WSDL extension definition

The requires element can be used as a subelement of the WSDL portType and operation elements. The element contains one or more intent names, as defined by the Policy Framework specification [SCA-POLICY]. Any service or reference that uses an interface marked with intents MUST implicitly add those intents to its own @requires list. [ASM80008]

SCA defines an attribute which is used to indicate that a given WSDL portType element (WSDL 1.1) has an associated callback interface. This is the @callback attribute, which applies to a WSDL portType element.
Snippet 6-5 shows the definition of the @callback attribute:

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

Snippet 6-5: callback WSDL extension definition

The value of the @callback attribute is the QName of a portType. The portType declared by the @callback attribute is the callback interface to use for the portType which is annotated by the @callback attribute.
Snippet 6-6 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>
```

Snippet 6-6: Example use of @callback

6.6 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 portType with the arguments and return of the service operations described using XML schema.

A WSDL interface is declared by an ***interface.wsdl*** element. Snippet 6-7 shows the pseudo-schema for the interface.wsdl element:

```
<!-- WSDL Interface schema snippet -->
<interface.wsdl interface="xs:anyURI" callbackInterface="xs:anyURI"?
                remotable="xs:boolean"?
                requires="listOfQNames"?
                policySets="listOfQNames">
   <requires/>*
   <policySetAttachment/>*
</interface.wsdl>
```

Snippet 6-7: interface.wsdl Pseudo-Schema

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

- ***interface : uri (1..1)*** - the URI of a WSDL portType

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

- ***callbackInterface : uri (0..1)*** - a callback interface, which is the URI of a WSDL portType

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

- ***remotable : boolean (0..1)*** – indicates whether the interface is remotable or not. @remotable has a default value of true. WSDL interfaces are always remotable and therefore an <interface.wsdl/> element MUST NOT contain remotable=”false”. [ASM80017]
- ***requires : listOfQNames (0..1)*** – a list of policy intents. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.
- ***policySets : listOfQNames (0..1)*** – a list of policy sets. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.

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

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

- ***requires : requires (0..n)*** - A service element has ***zero or more requires subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.
- ***policySetAttachment : policySetAttachment (0..n)*** - A service element has ***zero or more policySetAttachment subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.

6.6.1 Example of interface.wsdl

Snippet 6-8 shows an interface defined by the WSDL portType "StockQuote" with a callback interface defined by the "StockQuoteCallback" portType.

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

Snippet 6-8: Example interface.wsdl

7 Binding

Bindings are used by services, references, channels, consumers, and producers. 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. Producers, consumers and channels use bindings to describe the mechanism used to send and receive events.

SCA supports the use of multiple different types of bindings. Examples include ***SCA service, Web service, stateless session EJB, database stored procedure, EIS service***. 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, a reference, a channel, a consumer, or a producer element in a composite. Snippet 7-1 shows the composite pseudo-schema with the pseudo-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"?>*
         <wireFormat/>?
         <operationSelector/>?
         <requires/>*
         <policySetAttachment/>*
      </binding>
      <callback>?
         <binding uri="xs:anyURI"? name="xs:NCName"?
             requires="list of xs:QName"?
             policySets="list of xs:QName"?>+
             <wireFormat/>?
             <operationSelector/>?
             <requires/>*
             <policySetAttachment/>*
         </binding>
      </callback>
   </service>
   ...
   <reference ... >*
      <interface … />?
      <binding uri="xs:anyURI"? name="xs:NCName"?
         requires="list of xs:QName"?
         policySets="list of xs:QName"?>*
         <wireFormat/>?
         <operationSelector/>?
         <requires/>*
         <policySetAttachment/>*
      </binding>
      <callback>?
          <binding uri="xs:anyURI"? name="xs:NCName"?
             requires="list of xs:QName"?
             policySets="list of xs:QName"?>+
             <wireFormat/>?
             <operationSelector/>?
             <requires/>*
             <policySetAttachment/>*
         </binding>
      </callback>
   </reference>
   ...

   <channel … >
      <filters/>?
      <binding uri="xs:anyURI"? name="xs:NCName"?
         requires="list of xs:QName"?
         policySets="list of xs:QName"?>*
         <requires/>*
         <policySetAttachment/>*
         <filters/>*
      </binding>?
      <requires/>*
      <policySetAttachment/>*
   </channel>*

   <consumer … >
      <filters/>?
      <binding uri="xs:anyURI"? name="xs:NCName"?
         requires="list of xs:QName"?
         policySets="list of xs:QName"?>*
         <requires/>*
         <policySetAttachment/>*
         <filters/>*
      </binding>*
      <requires/>*
      <policySetAttachment/>*
   </consumer>*

   <producer … >
      <eventType/>?
      <binding uri="xs:anyURI"? name="xs:NCName"?
         requires="list of xs:QName"?
         policySets="list of xs:QName"?>*
         <requires/>*
         <policySetAttachment/>*
         <filters/>*
      </binding>*
      <requires/>*
      <policySetAttachment/>*
   </producer>*

</composite>
```

Snippet 7-1: composite Pseudo-Schema with binding Child element

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.sca, binding.ws, binding.ejb, binding.eis).

A ***binding*** element has the attributes:

- ***uri (0..1) -*** has the 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/bindingName 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 "Specifying the Target Service(s) for a Reference."
  - For a binding of a ***service*** the @uri attribute defines the bindingURI. If present, the bindingURI can be used by the binding as described in the section "Form of the URI of a Deployed 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, all non-callback bindings of the service or reference MUST have unique names, and all callback bindings of the service or reference MUST have unique names. [ASM90002] This uniqueness requirement implies that only one non-callback binding of a service or reference can have the default @name value, and only one callback binding of a service or reference can have the default @name value.

  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 [SCA-JMSBINDING] for examples of this referencing).
- ***requires (0..1)*** - a list of policy intents. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.
- ***policySets (0..1)*** – a list of policy sets. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.

A ***binding*** element has the child elements:

- ***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
- ***requires : requires (0..n)*** - A service element has ***zero or more requires subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.
- ***policySetAttachment : policySetAttachment (0..n)*** - A service element has ***zero or more policySetAttachment subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.
- ***filters : Filters (0..1)*** – a binding-specific ***zero or more filters subelement***. For more details on the filters subelement see Section Filters: Selecting Subsets of Events. The filters subelement of the bindings element contains filters that are binding-specific.

When multiple bindings exist for a service, channel, consumer, or producer, it means that the service, channel, consumer, or producer is available through any of the specified bindings. The technique that the SCA runtime uses to choose among available bindings is left to the implementation and it might include additional (nonstandard) configuration. Whatever technique is used needs to be documented by the runtime.

Services, References, consumers, and producers 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 wire to a specific binding of a target service the syntax "componentName/serviceName/bindingName" MUST be used. [ASM90004]

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

7.1 Messages containing Data not defined in the Service Interface

It is possible for a message to include information that is not defined in the interface used to define the service, for instance information can 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.

7.2 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 can 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 can include a definition of the wireFormat to use. This is done using an <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.

7.3 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 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.

7.4 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 a @uri attribute, which is termed a bindingURI.

If the bindingURI is specified on a given <binding/> element, the binding can 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.

7.4.1 Non-hierarchical URIs

Bindings that use non-hierarchical URI schemes (such as jms: or mailto:) can 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.

7.4.2 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 can 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 or port). 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 is worth noting that an intent supported by a binding type can completely change the behavior of the binding. For example, when the intent "confidentiality/transport" is attached to an HTTP binding, SSL is turned on. This basically changes the URI scheme of the binding from "http" to "https".

7.5 SCA Binding

The SCA binding element is defined by the pseudo-schema in Snippet 7-2.

```
<binding.sca uri="xs:anyURI"?
      name="xs:NCName"?
      requires="list of xs:QName"?
      policySets="list of xs:QName"?>
   <wireFormat/>?
   <operationSelector/>?
   <requires/>*
   <policySetAttachment/>*
   <filters/>?
</binding.sca>
```

Snippet 7-2: binding.sca pseudo-schema

A ***binding.sca*** element has the attributes:

- ***uri (0..1) -*** has the semantic:
  - The @uri attribute can be omitted.
  - If a <binding.sca/> element of a component reference specifies a URI via its @uri attribute, then this provides a wire to a target service provided by another component. The form of the URI which points to the service of a component that is in the same composite as the source component is as follows:

    <component-name>/<service-name>

    or

    <component-name>/<service-name>/<binding-name>

    in cases where the service has multiple bindings present.
  - The circumstances under which the @uri attribute can be used are defined in the section "Specifying the Target Service(s) for a Reference."
  - For a binding.sca of a component service, the @uri attribute MUST NOT be present. [ASM90005]
- ***name (0..1)*** – a name for the binding instance (an NCName), as defined for the base <binding/> element type.
- ***requires (0..1)*** - a list of policy intents. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.
- ***policySets (0..1)*** – a list of policy sets. See the Policy Framework specification [SCA-POLICY] for a description of this attribute.

A ***binding.sca*** element has the child elements:

- ***wireFormat (0..1)*** - a wireFormat to apply to the data flowing using the binding. binding.sca does not define any specific wireFormat elements.
- ***operationSelector(0..1)*** - an operationSelector element that is used to match a particular message to a particular operation in the interface. binding.sca does not define any specific operationSelector elements.
- ***requires : requires (0..n)*** - A service element has ***zero or more requires subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.
- ***policySetAttachment : policySetAttachment (0..n)*** - A service element has ***zero or more policySetAttachment subelements***. See the Policy Framework specification [SCA-POLICY] for a description of this element.
- ***filters: Filters (0..1) –*** binding-specific filters on event instances. This subelement can only be used for bindings specified on channels, consumers and producers.

The SCA binding can be used for (1) service interactions between references and services contained within the SCA Domain, (2) event interaction between consumers and channels, and (3) event interactions between producers and channels. 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 any specified qualities of service are implemented for the SCA binding type.

Qualities of service for <binding.sca/> are expressed using intents and/or policy sets following the rules defined in the Policy Framework specification [SCA-POLICY].

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 is used.

An SCA runtime has to support the binding.sca binding type. See the section on SCA Runtime conformance.

A service definition with no binding element specified uses the SCA binding (see ASM50005 in section 4.2 on Component Service). <binding.sca/> only has to be specified explicitly in override cases, or when a set of bindings is specified on a service definition and the SCA binding needs to be one of them.

If a reference does not have a binding subelement specified, then the binding used is one of the bindings specified by the service provider, as long as the intents attached to the reference and the service are all honoured, as described in the section on Component References.

A channel, producer, or consumer with no binding element specified uses the SCA binding.

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 <binding.sca/> element of a <component/> <reference/> specifies a URI via its @uri attribute, then this provides a wire to a target service provided by another component.

The form of the URI which points to the service of a component that is in the same composite as the source component is as follows:

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

7.5.1 Example SCA Binding

Snippet 7-3 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/200912"
                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>
```

Snippet 7-3: Example binding.sca

7.6 Web Service Binding

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

7.7 JMS Binding

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

8 Representation of Events and Event Types

Events in SCA MAY have an ***event type*** associated with them. Each event type is identified by a unique ***event type QName***.

An event can have no event type metadata associated with it - for example, this can be the case for events which are created by pre-existing non-SCA event sources.

SCA has a canonical ***representation*** of events in terms of XML Infoset and of event shapes in terms of XML Schema (see Section Event Type Definition Language Used by SCA). SCA event shapes are ***describable*** using XML Schema, although they don't have to be described using XML Schema – other type systems can be used. SCA events can have a wire format that is not XML.

Events can also have programming language specific representations. The details of the mapping between language specific formats and XML infoset are defined by the SCA implementation language specifications.

8.1 Event Type and Metadata

In SCA, ***event type*** definition consist of the following:

1. a unique ***event type QName***
2. a set of business data. This data is also called the ***shape*** of the event. It is possible that the same shape is used by multiple event types.
3. optional additional metadata associated with events of this type, such as creation time, and is separate from the event business data.

The shape of the event is defined in terms of an existing type system. Examples include XSD and Java.

For event shape defined using XSD, this is done in terms of an XML global element declaration.

8.2 Event Type Definition Language Used by SCA

SCA uses Web Services Event Descriptions (WS-EventDescriptions) **[WS-ED]** as its interoperable event type definition language. Any event type definition used in SCA MUST be mappable to WS-EventDescriptions. [ASM16001].

Snippet 8-1 provides an example of an event type definition.

```
<EventDescriptions xmlns="http://www.w3.org/2011/03/ws-evd"
             xmlns:foo="http://example.org/edl"
             targetNamespace="http://example.org/edl">
    <types>
        . . .
    </types>
    <eventType id="PrinterEvent" element="foo:PrinterEvent"/>
</EventDescriptions>
```

Snippet 8-1: Canonical Event Type Definition

8.3 Events with No Event Type

Events MAY have no event type metadata associated with them.

From an SCA perspective (and in particular, when dealing with events of this kind in Filter statements), such events are given the special event type name of sca:NULL (a QName). This special event type name MUST NOT be used in event instances for its type metadata. It is reserved for use in composite (such as in type filters on consumer, channels and type declarations on producers) to identify event instances that do not have any type metadata.

9 Filters: Selecting Subsets of Events

Event filters are used to select subsets of events from an event source. Event filters can be specified on consumers and on channels, and are then applied to the event instances that would otherwise be received by those consumers or channels.

Filters can operate against various sorts of data relating to an event instance:

- Event types
- Event business data
- Other event metadata

The mechanism for expressing filters is extensible, so that in the future filters can be added that operate against other data, such as Properties of the Event channel.

Filters can be expressed in a variety of dialects of filter language. It is possible to use different filter language dialects for different types of data - eg Event Metadata vs Business Data. It is possible to specify multiple filters (of the same type or different types) on a single consumer or channel.

Each filter expression must resolve to a boolean where "false" means that the event instance is discarded and "true" means that the event instance is passed by the filter. Where multiple filters are present, they are logically "AND"ed together so that only messages that pass all of the filters are passed by the collection of filters.

Filters can be specified on a channel, component consumer, composite consumer, or a consumer in the Component Type of an implementation. All filter expressions specified on a consumer, regardless of where (Component Type, Component or Composite) they are specified are logically “AND”ed together.

Filters have no side effects and filters have no state. They are evaluated against a particular event instance and indicate whether the event passes the filter or not – there are no other implications. This means that the order in which multiple filters are applied does not matter – the same result occurs whatever the order.

9.1 Form of Explicit Filter Elements

Explicit filters can be attached to various elements in SCA, such as consumers and channels. The syntax used to express the filters conveys three things:

1. The type of data that the filter operates against (the “subject”)
2. The language used to express the filter (the “dialect”)
3. The filter expression itself.

The choice of dialect might be constrained by the choice of subject; there are some dialect/subject combinations that do not make sense.

The filters, if any, that are attached to a consumer of channel are all contained in a single <sca:filters> element. The filters themselves MUST appear as child element of <sca:filters> and any element that is included as a child element of <sca:filters> MUST be a filter. The QName of the element indicates the subject of the filter and its dialect; SCA provides element declarations for all the filter subjects that it defines.

The element content is used to convey the expression, and is constrained by the dialect chosen.

The SCA specification defines a number of predefined filter subject/dialect elements. These are described in the following sections, but are summarized in the pseudo-schema in Snippet 9-1.

```
<filters>
   <eventType.sca qnames="list of xs:QName"? namespaces="list of xs:anyURI"?
/>*
    <body.xpath1> xs:string </body.xpath1>*
   <any>*
</filters> ?
```

Snippet 9-1: filters Pseudo-Schema

Note that the event filters are extensible, allowing new filter types to be defined as an extension and to be used at the place that the <any/> subelement is shown in the pseudo-schema. An SCA runtime is not required to support filter types not defined by this specification - but if an extended filter type is declared within a <filters/> element and the SCA runtime does not support that extended filter type, then the SCA runtime MUST generate an error when it encounters the declaration. [ASM17001]

9.2 Event Type Filters

Event type filters filter events based on the Event Type metadata of the event.

Only one dialect is currently defined for event type with the element name <eventType.sca>

```
<filters>
   <eventType.sca qnames="list of xs:QName"? namespaces="list of xs:anyURI"?
/>*
   ...
</filters>
```

Snippet 9-2: Pseudo-Schema for Event Type filter

In this dialect, a filter expression consists of either a list of one or more QNames specified as a value of the attribute @qname or a list of one of more namespace URIs specified as a value of the attribute @namespaces or both. This dialect filters on event types described using the WS-EventDescriiptions **[WS-ED]** specification as described in Section Event Type Definition Language Used by SCA.

Each QName in the list MUST be associated with a Namespace URI. This association is performed using the namespace declarations that are in-scope where the QName expression appears (e.g. in the composite document containing sca:Filter element). Unprefixed QNames are permitted, provided there is a default namespace declaration in-scope where the QName expression appears. QNames that belong to no namespace are not allowed.

A filter expressed in this dialect returns true if and only if either of the following is true:

1. at least one of the QNames specified in the @qnames attribute matches the QName of the event's Event Type. In order for a match to occur both these conditions must be true:
   - The associated Namespace URI's must contain an identical sequence of characters when expressed as Unicode code points.
   - The local parts of each QName must contain an identical sequence of characters when expressed as Unicode code points.
2. at least one of the namespaces specified in the @namespace attribute matches the namespace of the event's Event Type.
   - The Namespace URI's must contain an identical sequence of characters when expressed as Unicode code points.

9.2.1 Use of <eventType> on a Producer

The element <eventType> (and by extension <eventType.sca>) can also be used on a component type producer, component producer or a composite producer to declare the event types produced by the producer.

9.2.1.1 Use of <eventType.sca> on a Component Producer

When the element <eventType.sca> is specified on a component producer and the @qnames attribute is omitted, the value defaults to the value of the @qnames attribute for the producer of the same name in the componentType of the implementation used by the component. If the @qnames attribute is omitted and if the corresponding producer in the componentType of the implementation also does not have this attribute, then the producer is unconstrainted with respect to the Event Type QNames for the events that are sent by the producer. If the componentType has a value for @typeNames then the value of @typeNames for the component producer element MUST match that in the componentType. [ASM17002]

When the element <eventType.sca> is specified on a component producer and the @namespaces attribute is omitted, the value defaults to the value of the @namespaces attribute for the producer of the same name in the componentType of the implementation used by the component. If the @namespaces attribute is omitted, and if the corresponding producer in the componentType of the implementation also does not have this attribute, then the producer is unconstrainted with respect to the Even Type Namespace URI for the events that are sent by the producer. If the componentType has a value for @namespaces then the value of @namespaces for the component producer element MUST match that in the componentType. [ASM17003]

Note that both attributes @qnames and @namespaces can be used together. If both attributes are specified, then the component producer declares that it might send events whose Event Type is either listed in the @names attribute or whose Event Type belongs to one of the Namespaces listed in the @namespaces attribute.

9.2.1.2 Use of <eventType.sca> on a Composite Producer

When the element <eventType.sca> is specified on a component producer and the @qnames attribute is omitted, the value defaults to the value of the @qnames attribute, if present, of the associated component producer or the componentType of the component producer that is promoted.

If the associated component producer or the componentType producer has a value for @qnames then the value of @qames, if present, for the composite producer element MUST match that in the component propducer or the componentType producer. [ASM17004]

If the @namespaces attribute is omitted, the value defaults to the value of the @namespaces attribute, if present, of the associated component producer or the componentType of the component producer that is promoted.

If the associated component producer or the componentType producer has a value for @namespaces then the value of @namespaces, if present, for the composite producer element MUST match that in the component producer or the componentType producer.

Note that both attributes @qnames and @namespaces can be used together. If both attributes are specified, then the producer declares that it might send events whose Event Type is either listed in the @qnames attribute or whose Event Type belongs to one of the Namespaces listed in the @namespaces attribute.

9.2.2 Event Type Filter Examples

A filter that expresses interest in the events of types ns1:printer or ns2:printer:

<eventType.sca qnames="ns1:printer ns2:printer" />

A filter that expresses interest in events that do not have a type metadata:

<eventType.sca qnames="sca:NULL" />

A filter that expresses interest in events that either do not have type metadata or are of type ns1:printer:

<eventType.sca qnames="sca:NULL ns1:printer" />

A filter that expresses interest in events whose type belongs to one of the two namespaces "http://example.org/ns1" or "http://example.org/ns2":

<eventType.sca namespaces="http://example.org/ns1 http://example.org/ns2" />

A filter that expresses interest in events whose type belongs to the namespaces http://example.org/ns2 or is of type ns1:printer or is untyped:

```
<eventType.sca qnames="ns1:printer sca:NULL"
        namespaces="http://example.org/ns2" />
```

9.3 Business Data Filters

Business data filters filter events based on the business data contained within the event.

The following dialects are defined - xpath1.

9.3.1 XPATH 1.0 Dialect

Filter element QName: <sca:body.xpath1>

The Filter expression (content of the element <body.xpath1>) is an XPath 1.0 expression (not a predicate) whose context is:

- Context Node: the root element of the document being searched based upon the subject. In this case (the Business Data Subject) it is the root element of the event business data.
- Context Position: 1
- Context Size: 1
- Variable Binding: None
- Function Libraries: Core function library
- Namespace Declarations: Any namespace declarations in-scope where the XPath expression appears (e.g. in the SCDL document containing sca:Filter element)

This XPath expression can evaluate to one of four possible types: a node-set, a boolean, a number or a string. These result types are converted to a boolean value as follows:

- Node-set – false if no nodes, true otherwise
- boolean – no conversion
- string – false is empty string, true otherwise
- number – false if 0, true otherwise

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, implementation type definitions, and external attachment 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). An SCA runtime MUST make available to the Domain all the artifacts contained within the definitions.xml files in the Domain. [ASM10002] An SCA runtime MUST reject a definitions.xml file that does not conform to the sca-definitions.xsd schema. [ASM10003]

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 pseudo-schema shown in Snippet 10-1:

```
<?xml version="1.0" encoding="ASCII"?>
<!-- Composite schema snippet -->
<definitions    xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
                targetNamespace="xs:anyURI">

   <sca:intent/>*

   <sca:policySet/>*

   <sca:bindingType/>*

   <sca:implementationType/>*

</definitions>
```

Snippet 10-1: definitions Pseudo-Schema

The definitions element has the attribute:

- ***targetNamespace (1..1)*** – the namespace into which the child elements of this definitions element are placed (used for artifact resolution)

The definitions element contains child elements – intent, policySet, bindingType, implementationType and externalAttachmen. These elements are described elsewhere in this specification or in the Policy Framework specification [SCA-POLICY].

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 five XML Schema substitution group heads defined in the SCA namespace: ***interface***, ***implementation***, ***binding***, ***import*** and ***export*** for interface types, implementation types, binding types, import types and export types, respectively.

The SCA Client and Implementation specifications and the SCA Bindings specifications (see [SCA-COMMON-JAVA], [SCA-JAVA], [SCA BPEL], [SCA-CPP-Client], [SCA-C-Client], [SCA-WSBINDING], [SCA-JMSBINDING] as examples) use these XML Schema substitution groups to define some basic types of interfaces, implementations and bindings, but additional types can be defined as needed, where support for these extra ones is available from the runtime. The inteface type elements, implementation type elements, binding type elements, import type elements and export type elements defined by the SCA specifications are all part of the SCA namespace ("http://docs.oasis-open.org/ns/opencsa/sca/200912"), 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.

A conforming implementation type, interface type, import type or export type MUST meet the requirements in "Implementation Type Documentation Requirements for SCA Assembly Model Version 1.2 Specification". [ASM11001]

A binding extension element MUST be declared as an element in the substitution group of the sca:binding element. [ASM11002] A binding extension element MUST be declared to be of a type which is an extension of the sca:Binding type. [ASM11003]

11.1 Defining an Interface Type

Snippet 11-1 shows the base definition for the ***interface*** element and ***Interface*** type contained in ***sca-core.xsd***; see sca-core.xsd for the 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/200912"
        xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200912"
        elementFormDefault="qualified">

   ...

    <element name="interface" type="sca:Interface" abstract="true"/>
    <complexType name="Interface" abstract="true">
       <choice minOccurs="0" maxOccurs="unbounded">
          <element ref="sca:requires"/>
          <element ref="sca:policySetAttachment"/>
       </choice>
       <attribute name="remotable" type="boolean" use="optional"/>
       <attribute name="requires" type="sca:listOfQNames" use="optional"/>
       <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
    </complexType>

   ...
```

```

</schema>
```

Snippet 11-1: interface and Interface Schema

Snippet 11-2 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/200912"
        xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200912">

   <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>
```

Snippet 11-2: Extending interface to interface.java

Snippet 11-3 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/200912"
        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>
```

Snippet 11-3: Example interface extension

11.2 Defining an Implementation Type

Snippet 11-4 shows the base definition for the ***implementation*** element and ***Implementation*** type contained in ***sca-core.xsd***; see sca-core.xsdfor 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/200912"
        xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200912"
        elementFormDefault="qualified">

   ...

   <element name="implementation" type="sca:Implementation"
        abstract="true"/>
   <complexType name="Implementation" abstract="true">
      <complexContent>
         <extension base="sca:CommonExtensionBase">
           <choice minOccurs="0" maxOccurs="unbounded">
              <element ref="sca:requires"/>
              <element ref="sca:policySetAttachment"/>
           </choice>
            <attribute name="requires" type="sca:listOfQNames"
                       use="optional"/>
            <attribute name="policySets" type="sca:listOfQNames"
                       use="optional"/>
         </extension>
      </complexContent>
   </complexType>

   ...

</schema>
```

Snippet 11-4: implementation and Implementation Schema

Snippet 11-5 shows 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/200912"
        xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200912">

<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>
```

Snippet 11-5: Extending implementation to implementation.java

Snippet 11-6 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/200912"
        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>
```

Snippet 11-6: Example implementation extension

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 Snippet 11-7:

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

Snippet 11-7: implementationType Pseudo-Schema

The implementation type has the 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 [SCA-POLICY] for details.
- ***mayProvide (0..1)*** – a set of intents which the implementation type provides only when the intent is attached to the implementation element. See the Policy Framework specification [SCA-POLICY] for details.

11.3 Defining a Binding Type

Snippet 11-8 shows the base definition for the ***binding*** element and ***Binding*** type contained in ***sca-core.xsd***; see sca-core.xsdfor complete schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- binding type schema snippet -->
<!-- (c) Copyright SCA Collaboration 2006, 2009 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912"
        xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200912"
        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>
```

Snippet 11-8: binding and Binding Schema

Snippet 11-9 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/200912"
        xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200912">

   <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>
```

Snippet 11-9: Extending binding to binding.ws

Snippet 11-10 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/200912"
        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>
```

Snippet 11-10: Example binding extension

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 Snippet 11-11:

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

Snippet 11-11: bindingType Pseudo-Schema

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 [SCA-POLICY] for details.
- ***mayProvide (0..1)*** – a set of intents which the binding type provides only when the intent is attached to the binding element. See the Policy Framework specification [SCA-POLICY] for details.

11.4 Defining an Import Type

Snippet 11-12 shows the base definition for the ***import*** element and ***Import*** type contained in ***sca-core.xsd***; see sca-core.xsdfor complete schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright(C) OASIS(R) 2005,2009. 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/200912"
   targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912"
   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>
```

Snippet 11-12: import and Import Schema

Snippet 11-13 shows 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/200912"
        xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200912">

   <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>
```

Snippet 11-13: Extending import to import.java

Snippet 11-14 shows 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/200912"
        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>
```

Snippet 11-14: Example import extension

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

11.5 Defining an Export Type

Snippet 11-15 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) 2005,2009. 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/200912"
   targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912"
   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>
```

Snippet 11-15: export and Export Schema

Snippet 11-16 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/200912"
        xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200912">

   <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>
```

Snippet 11-16: Extending export to export.java

Snippet 11-17 we shows 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/200903"
        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>
```

Snippet 11-17: Example export extension

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

12 Packaging and Deployment

This section describes the SCA Domain and the packaging and deployment of artifacts contributed to the Domain.

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 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 are not able to tell that SCA is 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 might be the whole of a business, or it might 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 need 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 and definitions. XML artifacts that are not defined by SCA but which are 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 needed 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 might also be needed.

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 it is necessary for an SCA runtime to support the ZIP contribution format. 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 could 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 can also list namespaces of constructs that are defined within the contribution and which are available for use by other contributions, through export elements.

  These additional elements might not be physically present in the packaging, but might be generated based on the definitions and references that are present, or they might 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 is represented as a single artifact in the contribution hierarchy or whether all of the contents are represented as separate artifacts.

A goal of SCA's approach to deployment is that the contents of a contribution do 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, artifact-related or packaging-related artifact resolution 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 can be used 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 this 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 - e.g. the second contribution relates to a mainframe COBOL service whose interfaces are declared using a WSDL which is 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 (i.e. only a one-level search is performed).
2. from the contents of the contribution itself. [ASM12023]

Checking for errors in artifacts MUST NOT be done for artifacts in the Installed state (ie where the artifacts are simply part of installed contributions) [ASM12031]

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".

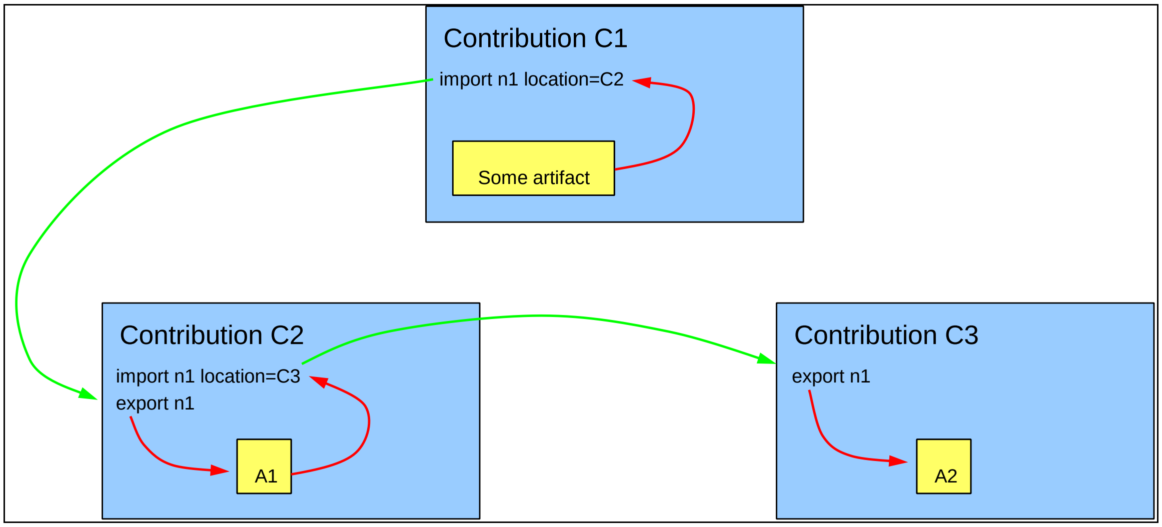


Figure 12-1: Example of SCA Artifact Resolution between Contributions

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 using artifact-related or packaging-related artifact resolution mechanisms, if present, by searching locations identified by the import statements of the contribution, if present, and by searching the contents of the contribution. [ASM12025]

12.2.2 SCA Contribution Metadata Document

The contribution can contain 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.

An SCA runtime MUST make the <import/> and <export/> elements found in the META-INF/sca-contribution.xml and META-INF/sca-contribution-generated.xml files available for the SCA artifact resolution process. [ASM12026] An SCA runtime MUST reject files that do not conform to the schema declared in sca-contribution.xsd. [ASM12027] An SCA runtime MUST merge the contents of sca-contribution-generated.xml into the contents of sca-contribution.xml, with the entries in sca-contribution.xml taking priority if there are any conflicting declarations. [ASM12028]

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/200912>

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

</contribution>
```

Snippet 12-1: contribution Pseudo-Schema

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. An SCA runtime MAY deploy the composites in <deployable/> elements found in the META-INF/sca-contribution.xml and META-INF/sca-contribution-generated.xml files. [ASM12029]

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 the export element:

- ***namespace (1..1)*** – For XML definitions, which are identified by QNames, the @namespace attribute of the export element MUST be the namespace URI for the exported definitions. [ASM12030] For XML technologies that define multiple *symbol spaces* that can be used within one namespace (e.g. WSDL portTypes are a different symbol space from WSDL bindings), all definitions from all symbol spaces are exported.

  Technologies that use naming schemes other than QNames 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 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 portTypes are a different symbol space from WSDL bindings), all definitions from all symbol spaces are imported.

  Technologies that use naming schemes other than QNames 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 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, it is recommended that dependent contributions are 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 a "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 definition of the ZIP file format is published by PKWARE in an Application Note on the .ZIP file format [ZIP-FORMAT].

12.3 States of Artifacts in the Domain

Artifacts in the SCA domain are in one of 3 states:

1. Installed
2. Deployed
3. Running

Installed artifacts are artifacts that are part of a Contribution that is installed into the Domain. Installed artifacts are available for use by other artifacts that are deployed, See "install Contribution" and "remove Contribution" to understand how artifacts are installed and uninstalled.

Deployed artifacts are artifacts that are available to the SCA runtime to be run.. Artifacts are deployed either through explicit deployment actions or through the presence of <deployable/> elements in sca-contribution.xml files within a Contribution. If an artifact is deployed which has dependencies on other artifacts, then those dependent artifacts are also deployed.

When the SCA runtime has one or more deployable artifacts, the runtime attempts to put those artifacts and any artifacts they depend on into the Running state. This can fail due to errors in one or more of the artifacts or the process can be delayed until all dependencies are available.

Checking for errors in artifacts MUST NOT be done for artifacts in the Installed state (ie where the artifacts are simply part of installed contributions) [ASM12032]

Errors in artifacts MUST be detected either during the Deployment of the artifacts, or during the process of putting the artifacts into the Running state, [ASM12033]

12.4 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 do not have to be provided 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 raise an error if there are conflicting names exported from multiple contributions. [ASM12007]

12.4.1 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 demands that any packaging format be able to offer up its artifacts in a single hierarchy).

12.5 Operations for Contributions

SCA Runtimes provide the following conceptual functionality associated with contributions to the Domain (meaning the function might not be represented as addressable services and also meaning that equivalent functionality might be provided in other ways). It is strongly encouraged that an SCA runtime provides the contribution operation functions (install Contribution, update Contribution, add Deployment Composite, update Deployment Composite, remove Contribution); how these are provided is implementation specific.

12.5.1 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 are 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.

12.5.2 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 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.

12.5.3 remove Contribution

Removes the deployed contribution identified by a supplied contribution URI.

12.6 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 (e.g. 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]

12.7 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 is implementation specific although it is expected that ONE of the 3 behaviours below is followed:

1) The SCA runtime disallows deployment of any components with autowire references. In this case, the SCA runtime can raise an exception at the point where the component is deployed.
2) The SCA runtime evaluates the target(s) for the reference at the time that the component is deployed and does not update those targets when later deployment actions occur.
3) The SCA runtime re-evaluates 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 reconfiguration of the reference takes place is described by the relevant client and implementation specifications.

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

12.7.1 add To Domain-Level Composite

This functionality adds the composite identified by a supplied URI to the Domain Level Composite. The supplied composite URI refers to a composite within an 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 composites components become top-level components and the component services become externally visible services (eg. they would be present in a WSDL description of the Domain). The meaning of any promoted services and references in the supplied composite is not defined; since there is no composite scope outside the domain composite, the usual idea of promotion has no utility.

12.7.2 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.

12.7.3 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 are resolved in terms of that installed composite.

12.7.4 get QName Definition

In order to make sense of the domain-level composite (as returned by get Domain-Level Composite), it needs to 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, e.g. 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 need to exist in some form, but not necessarily as a service operation with exactly this signature.

12.8 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 (i.e. this can 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 can have their references updated by the SCA runtime dynamically without the need to stop and start those components. How this is achieved is implementation specific.

Where components are updated by deployment actions (their configuration is changed in some way, which includes changing the wires of component references), the SCA implementation needs to ensure that the updates apply to all new instances of those components once the update is complete. An SCA runtime can choose to maintain existing instances with the old configuration of components updated by deployment actions, although an implementation of an SCA runtime can choose to stop and discard existing instances of those components.

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 can fail with a fault indicating that the service is unavailable. If the wire is the result of the autowire process, the SCA runtime can attempt to update the wire if there exists an alternative target component that satisfies the autowire process.

Where a component that is the target of a wire is updated, an SCA runtime can direct future invocations of that reference to the updated component.

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 can:

- 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.

12.9 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 defer the updating of the component property value until the component is stopped and restarted

[ASM12034]

13 SCA Runtime Considerations

This section describes aspects of an SCA Runtime that are defined by this specification.

13.1 Error Handling

The SCA Assembly specification identifies situations where the configuration of the SCA Domain and its contents are in error. When one of these situations occurs, the specification requires that the SCA Runtime that is interacting with the SCA Domain and the artifacts it contains recognises that there is an error, raise the error in a suitable manner and also refuse to run components and services that are in error.

The SCA Assembly specification is not prescriptive about the functionality of an SCA Runtime and the specification recognizes that there can be a range of design points for an SCA runtime. As a result, the SCA Assembly specification describes a range of error handling approaches which can be adopted by an SCA runtime.

An SCA Runtime MUST raise an error for every situation where the configuration of the SCA Domain or its contents are in error. The error is either raised at deployment time or at runtime, depending on the nature of the error and the design of the SCA Runtime. [ASM14005]

13.1.1 Errors which can be Detected at Deployment Time

Some error situations can be detected at the point that artifacts are deployed to the Domain. An example is a composite document that is invalid in a way that can be detected by static analysis, such as containing a component with two services with the same @name attribute.

An SCA runtime is expected to detect errors at deployment time where those errors can be found through static analysis. An SCA runtime has to prevent deployment of contributions that are in error, and raise an error to the process performing the deployment (e.g. write a message to an interactive console or write a message to a log file). The exact timing of checking contributions for errors is implementation specific.

The SCA Assembly specification recognizes that there are reasons why a particular SCA runtime finds it desirable to deploy contributions that contain errors (e.g. to assist in the process of development and debugging) - and as a result also supports an error handling strategy that is based on detecting problems at runtime. However, it is wise to consider reporting problems at an early stage in the deployment proocess.

13.1.2 Errors which are Detected at Runtime

An SCA runtime can detect problems at runtime. These errors can include some which can be found from static analysis (e.g. the inability to wire a reference because the target service does not exist in the Domain) and others that can only be discovered dynamically (e.g. the inability to invoke some remote Web service because the remote endpoint is unavailable).

Where errors can be detected through static analysis, the principle is that components that are known to be in error are not run. So, for example, if there is a component with a required reference (multiplicity 1..1 or 1..n) which is not wired, best practice is that the component is not run. If an attempt is made to invoke a service operation of that component, a "ServiceUnavailable" fault is raised to the invoker. It is also regarded as best practice that errors of this kind are also raised through appropriate management interfaces, for example to the deployer or to the operator of the system.

14 Conformance

The XML schema pointed to by the RDDL document at the namespace URI, defined by this specification, are considered to be authoritative and take 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, sca-interface-wsdl.xsd, sca-implementation-composite.xsd and sca-binding-sca.xsd schema.. [ASM13001]

An SCA runtime MUST reject a contribution file that does not conform to the sca-contribution.xsd schema. [ASM13002]

An SCA runtime MUST reject a definitions file that does not conform to the sca-definitions.xsd schema. [ASM13003]

There are two categories of artifacts that this specification defines conformance for: SCA Documents and SCA Runtimes.

14.1 SCA Documents

For a document to be a valid SCA Document, it MUST comply with one of the SCA document types below:

SCA Composite Document:

An SCA Composite Document is a file that MUST have an SCA <composite/> element as its root element and MUST conform to the sca-core-1.2.xsd schema and MUST comply with the additional constraints on the document contents as defined in Appendix C.

SCA ComponentType Document:

An SCA ComponentType Document is a file that MUST have an SCA <componentType/> element as its root element and MUST conform to the sca-core-1.2.xsd schema and MUST comply with the additional constraints on the document contents as defined in Appendix C.

SCA Definitions Document:

An SCA Definitions Document is a file that MUST have an SCA <definitions/> element as its root and MUST conform to the sca-definition-1.2.xsd schema and MUST comply with the additional constraints on the document contents as defined in Appendix C.

SCA Contribution Document:

An SCA Contribution Document is a file that MUST have an SCA <contributution/> element as its root element and MUST conform to the sca-contribution-1.2.xsd schema and MUST comply with the additional constraints on the document contents as defined in Appendix C.

SCA Interoperable Packaging Document:

A ZIP file containing SCA Documents and other related artifacts. The ZIP file SHOULD contain a top-level "META-INF" directory, and SHOULD contain a "META-INF/sca-contribution.xml" file, and MAY contain a "META-INF/sca-contribution-generated.xml" file.

14.2 SCA Runtime

An implementation that claims to conform to the requirements of an SCA Runtime defined in this specification MUST meet the following conditions:

1. The implementation MUST comply with all mandatory statements listed in table Mandatory Items in Appendix C: Conformance Items, related to an SCA Runtime.
2. The implementation MUST conform to the SCA Policy Framework v 1.1 Specification [SCA-POLICY].
3. The implementation MUST support at least one implementation type standardized by the OpenCSA Member Section or at least one implementation type that complies with the following rules:

   a. The implementation type is defined in compliance with the SCA Assembly Extension Model (Section 9 of the SCA Assembly Specification).

   b. A document describing the mapping of the constructs defined in the SCA Assembly specification with those of the implementation type exists and is made available to its prospective user community. Such a document describes how SCA components can be developed using the implementation type, how these components can be configured and assembled together (as instances of Components in SCA compositions). The form and content of such a document are described in the specification "Implementation Type Documentation Requirements for SCA Assembly Model Version 1.1 Specification" [SCA-IMPLTYPDOC]. The contents outlined in this specification template MUST be provided in order for an SCA runtime to claim compliance with the SCA Assembly Specification on the basis of providing support for that implementation type. An example of a document that describes an implementation type is the "SCA POJO Component Implementation Specification Version 1.1" [SCA-Java].

   c. An adapted version of the SCA Assembly Test Suite which uses the implementation type exists and is made available to its prospective user community. The steps required to adapt the SCA Assembly Test Suite for a new implementation type are described in the specification "Test Suite Adaptation for SCA Assembly Model Version 1.1 Specification" [SCA-TSA]. The requirements described in this specification MUST be met in order for an SCA runtime to claim compliance with the SCA Assembly Specification on the basis of providing support for that implementation type.
4. The implementation MUST support binding.sca and MUST support and conform to the SCA Web Service Binding Specification v 1.1 [SCA-WSBINDING].

14.2.1 Optional Items

In addition to mandatory items, Appendix C: Conformance Items lists a number of non-mandatory items that can be implemented SCA Runtimes. These items are categorized into functionally related classes as follows:

- Development – items to improve the development of SCA contributions, debugging, etc.
- Enhancement – items that add functionality and features to the SCA Runtime.
- Interoperation – items that improve interoperability of SCA contributions and Runtimes

These classifications are not rigid and some may overlap; items are classified according to their primary intent.

Appendix A. XML Schemas

A.1 sca.xsd

sca-1.2.xsd is provided for convenience. It contains <include/> elements for each of the schema files that contribute to the http://docs.oasis-open.org/ns/opencsa/sca/200912 namespace.

A.2 sca-core.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright(C) OASIS(R) 2005,2011. 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/200912"
   targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912"
   elementFormDefault="qualified">

   <include schemaLocation="sca-policy-1.1-cd03.xsd"/>
   <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"/>
                  <element name="consumer" type="sca:ComponentConsumer"/>
                  <element name="producer" type="sca:ComponentProducer"/>

               </choice>
               <any namespace="##other" processContents="lax" minOccurs="0"
                    maxOccurs="unbounded"/>
            </sequence>
         </extension>
      </complexContent>
   </complexType>

   <!-- Composite -->
   <element name="composite" type="sca:Composite"/>
   <complexType name="Composite">
      <complexContent>
         <extension base="sca:CommonExtensionBase">
            <sequence>
               <element ref="sca:include" minOccurs="0"
                        maxOccurs="unbounded"/>
               <choice minOccurs="0" maxOccurs="unbounded">
                  <element ref="sca:requires"/>
                  <element ref="sca:policySetAttachment"/>
                  <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="channel" type="sca:Channel"/>
                  <element name="consumer" type="sca:Consumer"/>
                  <element name="producer" type="sca:Producer"/>

                  <element name="wire" type="sca:Wire"/>
               </choice>
               <element ref="sca:extensions" minOccurs="0" maxOccurs="1"/>
            </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="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 ref="sca:binding" minOccurs="0"
                        maxOccurs="unbounded" />
               <element ref="sca:callback" minOccurs="0" maxOccurs="1" />
               <element ref="sca:requires" minOccurs="0"
                        maxOccurs="unbounded"/>
               <element ref="sca:policySetAttachment" minOccurs="0"
                        maxOccurs="unbounded"/>
               <element ref="sca:extensions" minOccurs="0" maxOccurs="1" />
            </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">
         <choice minOccurs="0" maxOccurs="unbounded">
            <element ref="sca:requires"/>
            <element ref="sca:policySetAttachment"/>
         </choice>
         <attribute name="remotable" type="boolean" use="optional"/>
      <attribute name="requires" type="sca:listOfQNames"
            use="optional"/>
      <attribute name="policySets" type="sca:listOfQNames"
            use="optional"/>
      </extension>
   </complexContent>
</complexType>

<!-- Reference -->
<complexType name="Reference">
   <complexContent>
      <extension base="sca:Contract">
         <attribute name="target" type="sca:listOfAnyURIs"
                    use="optional"/>
         <attribute name="wiredByImpl" type="boolean" use="optional"
                    default="false"/>
         <attribute name="multiplicity" type="sca:Multiplicity"
                    use="required"/>
         <attribute name="promote" type="sca:listOfAnyURIs"
                    use="required"/>
      </extension>
   </complexContent>
</complexType>

<complexType name="ConsumerContract">
   <complexContent>
      <extension base="sca:CommonExtensionBase">
         <sequence>
            <element ref="sca:filters" minOccurs="0" maxOccurs="1" />
            <element ref="sca:requires" minOccurs="0"
                     maxOccurs="unbounded"/>
            <element ref="sca:policySetAttachment" minOccurs="0"
                     maxOccurs="unbounded"/>
            <element ref="sca:extensions" minOccurs="0" maxOccurs="1" />
         </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>

<complexType name="Consumer">
   <complexContent>
```

```
         <extension base="sca:ConsumerContract">
            <sequence/>
            <attribute name="promote" type="sca:listOfAnyURIs"
                       use="required"/>
         </extension>
      </complexContent>
   </complexType>

   <complexType name="ProducerContract">
      <complexContent>
         <extension base="sca:CommonExtensionBase">
            <sequence>
               <element ref="sca:eventType" minOccurs="0" maxOccurs="1" />
               <element ref="sca:requires" minOccurs="0"
                        maxOccurs="unbounded"/>
               <element ref="sca:policySetAttachment" minOccurs="0"
                        maxOccurs="unbounded"/>
               <element ref="sca:extensions" minOccurs="0" maxOccurs="1" />
            </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>

   <complexType name="Producer">
      <complexContent>
         <extension base="sca:ProducerContract">
            <sequence/>
            <attribute name="promote" type="sca:listOfAnyURIs"
                       use="required"/>
         </extension>
      </complexContent>
   </complexType>

   <complexType name="Channel">
      <complexContent>
         <extension base="sca:CommonExtensionBase">
            <sequence>
               <element ref="sca:filters" minOccurs="0" maxOccurs="1" />
               <element ref="sca:binding" minOccurs="0"
                        maxOccurs="1" />
               <element ref="sca:requires" minOccurs="0"
                        maxOccurs="unbounded"/>
               <element ref="sca:policySetAttachment" minOccurs="0"
                        maxOccurs="unbounded"/>
               <element ref="sca:extensions" minOccurs="0" maxOccurs="1" />
            </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>

   <!-- Property -->
   <complexType name="SCAPropertyBase" mixed="true">
      <sequence>
         <any namespace="##any" processContents="lax" minOccurs="0"
```

```
              maxOccurs="unbounded"/>
         <!-- 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="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="value" type="anySimpleType" use="optional"/>
      <anyAttribute namespace="##other" processContents="lax"/>
   </complexType>

   <complexType name="Property" mixed="true">
      <complexContent mixed="true">
         <extension base="sca:SCAPropertyBase">
            <attribute name="mustSupply" type="boolean" use="optional"
                       default="false"/>
         </extension>
      </complexContent>
   </complexType>

   <complexType name="PropertyValue" mixed="true">
      <complexContent mixed="true">
         <extension base="sca:SCAPropertyBase">
            <attribute name="source" type="string" use="optional"/>
            <attribute name="file" type="anyURI" use="optional"/>
         </extension>
      </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 ref="sca:requires" minOccurs="0"
                        maxOccurs="unbounded"/>
               <element ref="sca:policySetAttachment" minOccurs="0"
                        maxOccurs="unbounded"/>
               <element ref="filters" minOccurs="0" maxOccurs="1"/>
            </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" abstract="true"/>
   <complexType name="WireFormatType" abstract="true">
      <anyAttribute namespace="##other" processContents="lax"/>
   </complexType>

   <!-- OperationSelector Type -->
   <element name="operationSelector" type="sca:OperationSelectorType"
      abstract="true"/>
   <complexType name="OperationSelectorType" abstract="true">
      <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"/>
               <element ref="sca:requires"/>
               <element ref="sca:policySetAttachment"/>
               <element ref="sca:extensions" minOccurs="0" maxOccurs="1"/>
            </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="1"
                  maxOccurs="1"/>
               <choice minOccurs="0" maxOccurs="unbounded">
                  <element name="service" type="sca:ComponentService"/>
                  <element name="reference" type="sca:ComponentReference"/>
                  <element name="property" type="sca:PropertyValue"/>
                  <element ref="sca:requires"/>
                  <element ref="sca:policySetAttachment"/>
                  <element name="consumer" type="sca:ComponentConsumer"/>
                  <element name="producer" type="sca:ComponentProducer"/>
               </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="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"/>
            <attribute name="nonOverridable" type="boolean" use="optional"
                       default="false"/>
         </extension>
      </complexContent>
   </complexType>

   <complexType name="ComponentConsumer">
      <complexContent>
         <extension base="sca:ConsumerContract">
            <sequence/>
            <attribute name="source" type="sca:listOfAnyURIs"
                       use="optional"/>
         </extension>
      </complexContent>
   </complexType>

   <complexType name="ComponentProducer">
      <complexContent>
         <extension base="sca:ProducerContract">
            <sequence/>
            <attribute name="target" type="sca:listOfAnyURIs"
                       use="optional"/>
         </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 ref="sca:binding" minOccurs="0"
                        maxOccurs="unbounded"/>
               <element ref="sca:callback" minOccurs="0"/>
               <element ref="sca:requires" minOccurs="0"
                        maxOccurs="unbounded"/>
               <element ref="sca:policySetAttachment" minOccurs="0"
                        maxOccurs="unbounded"/>
               <element ref="sca:extensions" minOccurs="0" maxOccurs="1" />
            </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">
         <choice minOccurs="0" maxOccurs="unbounded">
            <element ref="sca:requires"/>
            <element ref="sca:policySetAttachment"/>
         </choice>
            <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"/>
            <attribute name="replace" type="boolean" use="optional"
               default="false"/>
         </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>

   <!-- Extensions element -->
   <element name="extensions">
      <complexType>
         <sequence>
            <any namespace="##other" processContents="lax"
               minOccurs="1" maxOccurs="unbounded"/>
         </sequence>
      </complexType>
   </element>

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

   <!-- 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"/>

   <!-- Value type definition for property values -->
   <element name="value" type="sca:ValueType"/>
   <complexType name="ValueType" mixed="true">
      <sequence>
         <any namespace="##any" processContents="lax" minOccurs="0"
            maxOccurs='unbounded'/>
      </sequence>
      <!-- mixed="true" to handle simple type -->
      <anyAttribute namespace="##any" processContents="lax"/>
   </complexType>

   <!-- 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>

   <simpleType name="CreateResource">
      <restriction base="string">
         <enumeration value="always" />
         <enumeration value="never" />
         <enumeration value="ifnotexist" />
      </restriction>
   </simpleType>

    <element name="filters" type="sca:Filter"/>
    <complexType name="Filter">
        <sequence>
            <choice minOccurs="0" maxOccurs="unbounded">
                  <element ref="sca:eventType" />
                  <element ref="sca:body.xpath1" />
            </choice>
            <any namespace="##other" processContents="lax" minOccurs="0"
              maxOccurs="unbounded"/>
        </sequence>
        <anyAttribute namespace="##other" processContents="lax"/>
    </complexType>

    <element name="eventType" abstract="true"/>

    <element name="eventType.sca" type="sca:EventType.sca"
             substitutionGroup="eventType"/>

    <complexType name="EventType.sca">
        <sequence>
            <any namespace="##other" processContents="lax" minOccurs="0"
              maxOccurs="unbounded"/>
        </sequence>
        <attribute name="qnames" type="sca:listOfQNames" />
        <attribute name="namespaces" type="sca:listOfAnyURIs" />
        <anyAttribute namespace="##other" processContents="lax" />
    </complexType>

    <element name="body.xpath1" type="string" />

</schema>
```

A.3 sca-binding-sca.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright(C) OASIS(R) 2005,2011. 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/200912"
        xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200912"
        elementFormDefault="qualified">

   <include schemaLocation="sca-core-1.2-csd01.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

Is described in the SCA Java Common Annotations and APIs specification [SCA-Common-Java]

A.5 sca-interface-wsdl.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright(C) OASIS(R) 2005,2011. 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/200912"
   xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200912"
   elementFormDefault="qualified">

   <include schemaLocation="sca-core-1.2-csd01.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"/>
         </extension>
      </complexContent>
   </complexType>

</schema>
```

A.6 sca-implementation-java.xsd

Is described in the Java Component Implementation specification [SCA-Java]

A.7 sca-implementation-composite.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright(C) OASIS(R) 2005,2011. 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/200912"
   targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912"
   elementFormDefault="qualified">

   <include schemaLocation="sca-core-1.2-csd01.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-binding-webservice.xsd

Is described in the SCA Web Services Binding specification [SCA-WSBINDING]

A.9 sca-binding-jms.xsd

Is described in the SCA JMS Binding specification [SCA-JMSBINDING]

A.10 sca-policy.xsd

Is described in the Policy Framework specification [SCA-POLICY]

A.11 sca-contribution.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright(C) OASIS(R) 2005,2011. 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/200912"
   targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912"
   elementFormDefault="qualified">

   <include schemaLocation="sca-core-1.2-csd01.xsd"/>

   <!-- Contribution -->
   <element name="contribution" type="sca:ContributionType"/>
   <complexType name="ContributionType">
      <complexContent>
         <extension base="sca:CommonExtensionBase">
            <sequence>
               <element name="deployable" type="sca:DeployableType"
                        minOccurs="0" maxOccurs="unbounded"/>
               <element ref="sca:importBase" minOccurs="0"
                        maxOccurs="unbounded"/>
               <element ref="sca:exportBase" minOccurs="0"
                        maxOccurs="unbounded"/>
               <element ref="sca:extensions" minOccurs="0" maxOccurs="1"/>
            </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"
            substitutionGroup="sca:importBase"/>
   <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"
            substitutionGroup="sca:exportBase"/>
   <complexType name="ExportType">
      <complexContent>
         <extension base="sca:Export">
            <attribute name="namespace" type="string" use="required"/>
         </extension>
      </complexContent>
   </complexType>

</schema>
```

A.12 sca-definitions.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright(C) OASIS(R) 2005,2011. 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/200912"
   xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200912"
   elementFormDefault="qualified">

   <include schemaLocation="sca-core-1.2-csd01.xsd"/>
   <include schemaLocation="sca-policy-1.1-cd02.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:bindingType"/>
               <element ref="sca:implementationType"/>
               <element ref="sca:externalAttachment"/>
               <any namespace="##other" processContents="lax"
                  minOccurs="0" maxOccurs="unbounded"/>
            </choice>
            <attribute name="targetNamespace" type="anyURI" use="required"/>
         </extension>
      </complexContent>
   </complexType>

</schema>
```

Appendix 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 use to call the service.

SCA supports multiple different types of bindings. Examples include ***SCA service, Web service, stateless session EJB, database 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 runtime, vendors can choose to support the ones that are important for them. A single SCA implementation can 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 (e.g. public Web services for use by other organizations). The operations provided by a service are specified by an Interface, as are the operations needed by the service client (if there is one).   An implementation can contain multiple services, when it is possible to address the services of the implementation separately.

A service can 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.

Interfaces can be identified as remotable through the <interface /> XML, but are typically specified as remotable using a component implementation technology specific mechanism, such as Java annotations. 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 can rely on by-reference calling conventions, or can assume a very fine-grained interaction style that is incompatible with remote distribution. They can 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 provided 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 can 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 database 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 can be ***bi-directional***. A bi-directional service has service operations which are provided by each end of a service communication – this could be the case where a particular service demands a “callback” interface on the client, which it 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 can be used as a component implementation. When used in this way, it defines a boundary for Component visibility. Components cannot 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 can 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. A Domain also contains Wires that connect together the Components. A Domain does not contain promoted Services or promoted References, since promotion has no meaning at the Domain level.

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.

B.12 SCA Runtime

An SCA Runtime is a set of one or more software programs which, when executed, can accept and run SCA artifacts as defined in the SCA specifications. An SCA runtime provides an implementation of the SCA Domain and an implementation of capabilities for populating the domain with artifacts and with capabilities for running specific artifacts. An SCA Runtime can vary in size and organization and can involve a single process running on a single machine, multiple processes running on a single machine or multiple processes running across multiple machines that are linked by network communications.

An SCA runtime supports at least one SCA implementation type and also supports at least one binding type.

SCA Runtimes can include tools provided to assist developers in creating, testing and debugging of SCA applications and can be used to host and run SCA applications that provide business capabilities.

An SCA runtime can be implemented using any technologies (i.e. it is not restricted to be implemented using any particular technologies) and it can be hosted on any operating system platform.

B.13 Channel

TODO: cut-and-paste from main document once it is finalized.

B.14 Consumer

TODO: cut-and-paste from main document once it is finalized.

B.15 Producer

TODO: cut-and-paste from main document once it is finalized

B.16 Filters

TODO: cut-and-paste from main document once it is finalized

B.17 Event Types

TODO: cut-and-paste from main document once it is finalized

Appendix C. Conformance Items

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

Mandatory Items

Conformance ID	Description
[ASM40001]	The extension of a componentType side file name MUST be .componentType.
[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.
[ASM40010]	A single property element MUST NOT contain both a @type attribute and an @element attribute.
[ASM40011]	When the componentType has @mustSupply="true" for a property element, a component using the implementation MUST supply a value for the property since the implementation has no default value for the property.
[ASM40012]	The value of the property @file attribute MUST be a dereferencable URI to a file containing the value for the property.
[ASM40101]	The @name attribute of a <consumer/> child element of a <componentType/> MUST be unique amongst the consumer elements of that <componentType/>.
[ASM40102]	For artifacts that have service/consumer dual manifestation, the service interface MUST • be non-bidirectional • have only one-way operations
[ASM40103]	The @name attribute of a <producer/> child element of a <componentType/> MUST be unique amongst the producer

	elements of that <componentType/>.
[ASM40104]	For artifacts that have reference/producer dual manifestation, the reference interface MUST • be non-bidirectional • have only one-way operations
[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 an interface is declared for a component service, the interface MUST provide a compatible subset of the interface declared for the equivalent service in 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, the interface MUST provide a compatible superset of the interface

	declared for the equivalent reference in the componentType of the implementation.
[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. 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 uses more than a simple URI for the address of a target service. In cases where a reference element has a binding subelement that uses more than simple URI, the @uri attribute of the binding element MUST NOT be used to identify the target service - in this case binding specific attributes and/or child elements MUST be used.
[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 raise 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.
[ASM50031]	The @name attribute of a property element of a <component/> MUST be unique amongst the property elements of that <component/>.
[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.
[ASM50035]	A single property element MUST NOT contain both a @type attribute and an @element attribute.
[ASM50036]	The property type specified for the property element of a component MUST be compatible with the type of the property with the same @name declared in the component type of the implementation used by the component. If no type is declared in the component property element, the type of the property declared in the componentType of the implementation MUST be used.
[ASM50037]	The @name attribute of a property element of a <component/> MUST match the @name attribute of a property element of the componentType of the <implementation/> child element of the component.
[ASM50038]	In these cases where the types of two property elements are matched, the types declared for the two <property/> elements MUST be compatible
[ASM50039]	A reference with multiplicity 0..1 MUST have no more than one target service defined.
[ASM50040]	A reference with multiplicity 1..1 MUST have exactly one target service defined.
[ASM50041]	A reference with multiplicity 1..n MUST have at least one target service defined.
[ASM50042]	If a component reference has @multiplicity 0..1 or 1..1 and @nonOverridable==true, then the component reference MUST NOT be promoted by any composite reference.
[ASM50043]	The default value of the @autowire attribute MUST be the value of the @autowire attribute on the component containing the reference, if present, or else the value of the @autowire attribute of the composite containing the component, if present, and if neither is present, then it is "false".
[ASM50044]	When a property has multiple values set, all the values MUST be contained within a single property element.
[ASM50045]	The value of the component property @file attribute MUST be a

	dereferencable URI to a file containing the value for the property.
[ASM50046]	The format of the file which is referenced by the @file attribute of a component property or a componentType property is that it is an XML document which MUST contain an sca:values element which in turn contains one of: • a set of one or more <sca:value/> elements each containing a simple string - where the property type is a simple XML type • a set of one or more <sca:value/> elements or a set of one or more global elements - where the property type is a complex XML type
[ASM50101]	The @name attribute of a consumer element of a <component/> MUST be unique amongst the consumer elements of that <component/>.
[ASM50102]	The @name attribute of a consumer element of a <component/> MUST match the @name attribute of a consumer element of the componentType of the <implementation/> child element of the component.
[ASM50103]	The @name attribute of a producer element of a <component/> MUST be unique amongst the producer elements of that <component/>.
[ASM50104]	The @name attribute of a producer element of a <component/> MUST match the @name attribute of a producer element of the componentType of the <implementation/> child element of the component.
[ASM60001]	A composite @name attribute value 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.
[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 each of the component reference interfaces MUST be a compatible 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 multiplicity of a composite reference MUST be equal to or further restrict the multiplicity of each of the component references that it promotes, with the exception that the multiplicity of the composite reference does not have to require a target if there is already a target on the component reference. This means that a component reference with multiplicity 1..1 and a target can be promoted by a composite reference with multiplicity 0..1, and a component reference with multiplicity 1..n and one or more targets can be promoted by a composite reference with multiplicity 0..n or 0..1.
[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).
[ASM60013]	If no interface is declared on a composite reference, the interface from one of its promoted component references MUST be used for the component type associated with the composite.
[ASM60014]	The @name attribute of a composite property MUST be unique amongst the properties of the same composite.
[ASM60022]	For each component reference for which autowire is enabled, the SCA runtime MUST search within the composite for target services which have an interface that is a compatible superset of the interface of the reference.
[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.
[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.
[ASM60035]	All the component references promoted by a single composite reference MUST have the same value for @wiredByImpl.
[ASM60036]	If the @wiredByImpl attribute is not specified on the composite reference, the default value is "true" if all of the promoted component references have a wiredByImpl value of "true", and the default value is "false" if all the promoted component references have a wiredByImpl value of "false". If the @wiredByImpl attribute is specified, its value MUST be "true" if all of the promoted component references have a wiredByImpl value of "true", and its value MUST be "false" if all the promoted component references have a wiredByImpl value of "false".
[ASM60037]	<include/> processing MUST take place before the processing of the @promote attribute of a composite reference is performed.
[ASM60038]	<include/> processing MUST take place before the processing of the @promote attribute of a composite service is performed.
[ASM60039]	<include/> processing MUST take place before the @source and @target attributes of a wire are resolved.
[ASM60040]	A single property element MUST NOT contain both a @type attribute and an @element attribute.
[ASM60041]	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 an error.
[ASM60042]	The @name attribute of an include element MUST be the QName of a composite in the SCA Domain.
[ASM60043]	The interface declared by the target of a wire MUST be a compatible superset of the interface declared by the source of the wire.
[ASM60045]	An SCA runtime MUST introspect the componentType of a Composite used as a Component Implementation following the rules defined in the section "Component Type of a Composite used as a Component Implementation"
[ASM60046]	If <service-name> is present, the component service with @name corresponding to <service-name> MUST be used for the wire.

[ASM60047]	If there is no component service with @name corresponding to <service-name>, the SCA runtime MUST raise an error.
[ASM60048]	If <service-name> is not present, the target component MUST have one and only one service with an interface that is a compatible superset of the wire source’s interface and satisifies the policy requirements of the wire source, and the SCA runtime MUST use this service for the wire.
[ASM60049]	If <binding-name> is present, the <binding/> subelement of the target service with @name corresponding to <binding-name> MUST be used for the wire.
[ASM60050]	If there is no <binding/> subelement of the target service with @name corresponding to <binding-name>, the SCA runtime MUST raise an error.
[ASM60051]	If <binding-name> is not present and the target service has multiple <binding/> subelements, the SCA runtime MUST choose one and only one of the <binding/> elements which satisfies the mutual policy requirements of the reference and the service, and the SCA runtime MUST use this binding for the wire.
[ASM60101]	The name of the consumer MUST be unique amongst the consumer elements of the composite.
[ASM60102]	A composite <consumer/> element's @promote attribute MUST identify one of the component consumers within that composite.
[ASM60103]	<include/> processing MUST take place before the processing of the @promote attribute of a composite service is performed.
[ASM60104]	The name of the producer MUST be unique amongst the producer elements of the composite.
[ASM60105]	A composite <producer/> element's @promote attribute MUST identify one of the component producers within that composite.
[ASM60106]	<include/> processing MUST take place before the processing of the @promote attribute of a composite service is performed.
[ASM60107]	The name of the channel MUST be unique amongst the channel elements of the composite.
[ASM60108]	An SCA runtimes MUST support the use of domain channels.
[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.
[ASM80008]	Any service or reference that uses an interface marked with 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.
[ASM80016]	The interface.wsdl @callbackInterface attribute, if present, MUST reference a portType of a WSDL 1.1 document.
[ASM80017]	WSDL interfaces are always remotable and therefore an <interface.wsdl/> element MUST NOT contain remotable="false".
[ASM90001]	For a binding of a ***reference*** the @uri attribute defines the target URI of the reference. This MUST be either the componentName/serviceName/bindingName 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, all non-callback bindings of the service or reference MUST have unique names, and all callback bindings of the service or reference MUST have unique names.
[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]	To wire to a specific binding of a target service the syntax "componentName/serviceName/bindingName" MUST be used.
[ASM90005]	For a binding.sca of a component service, the @uri attribute MUST NOT be present.
[ASM10001]	all of the QNames for the definitions contained in definitions.xml files MUST be unique within the Domain.
[ASM10002]	An SCA runtime MUST make available to the Domain all the

	artifacts contained within the definitions.xml files in the Domain.
[ASM10003]	An SCA runtime MUST reject a definitions.xml file that does not conform to the sca-definitions.xsd schema.
[ASM11001]	A conforming implementation type, interface type, import type or export type MUST meet the requirements in "Implementation Type Documentation Requirements for SCA Assembly Model Version 1.2 Specification".
[ASM11002]	A binding extension element MUST be declared as an element in the substitution group of the sca:binding element.
[ASM11003]	A binding extension element MUST be declared to be of a type which is an extension of the sca:Binding type.
[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
[ASM12005]	Where present, artifact-related or packaging-related artifact resolution 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.
[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 (e.g. 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".
[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 (i.e. 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 using artifact-related or packaging-related artifact resolution mechanisms, if present, by searching locations identified by the import statements of the contribution, if present, and by searching the contents of the contribution.
[ASM12026]	An SCA runtime MUST make the <import/> and <export/> elements found in the META-INF/sca-contribution.xml and META-INF/sca-contribution-generated.xml files available for the SCA artifact resolution process.
[ASM12027]	An SCA runtime MUST reject files that do not conform to the schema declared in sca-contribution.xsd.
[ASM12028]	An SCA runtime MUST merge the contents of sca-contribution-generated.xml into the contents of sca-contribution.xml, with the entries in sca-contribution.xml taking priority if there are any conflicting declarations.
[ASM12030]	For XML definitions, which are identified by QNames, the @namespace attribute of the export element MUST be the namespace URI for the exported definitions.
[ASM12031]	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.
[ASM12032]	Checking for errors in artifacts MUST NOT be done for artifacts in the Installed state (ie where the artifacts are simply part of installed contributions)
[ASM12033]	Errors in artifacts MUST be detected either during the Deployment of the artifacts, or during the process of putting the artifacts into the Running state,
[ASM12034]	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 defer the updating of the component property value until the component is stopped and restarted
[ASM13001]	An SCA runtime MUST reject a composite file that does not conform to the sca-core.xsd, sca-interface-wsdl.xsd, sca-implementation-composite.xsd and sca-binding-sca.xsd schema.

[ASM13002]	An SCA runtime MUST reject a contribution file that does not conform to the sca-contribution.xsd schema.
[ASM13003]	An SCA runtime MUST reject a definitions file that does not conform to the sca-definitions.xsd schema.
[ASM14005]	An SCA Runtime MUST raise an error for every situation where the configuration of the SCA Domain or its contents are in error. The error is either raised at deployment time or at runtime, depending on the nature of the error and the design of the SCA Runtime.
[ASM16001]	Any event type definition used in SCA MUST be mappable to WS-EventDescriptions.
[ASM17001]	An SCA runtime is not required to support filter types not defined by this specification - but if an extended filter type is declared within a <filters/> element and the SCA runtime does not support that extended filter type, then the SCA runtime MUST generate an error when it encounters the declaration.
[ASM17002]	If the componentType has a value for @typeNames then the value of @typeNames for the component producer element MUST match that in the componentType.
[ASM17003]	If the componentType has a value for @namespaces then the value of @namespaces for the component producer element MUST match that in the componentType.
[ASM17004]	If the associated component producer or the componentType producer has a value for @qnames then the value of @qames, if present, for the composite producer element MUST match that in the component propducer or the componentType producer.

C.1 Non-mandatory Items

Conformance ID	Description	Classification
[ASM20101]	A producer SHOULD only produce event type it has declared.	Interoperation
[ASM20102]	An SCA Runtime MAY reject events of a type from a producer which does not declare that it produces events of that type.	Interoperation
[ASM60109]	An SCA runtime MAY support the use of private channels.	Enhancement
[ASM12002]	Within any contribution packaging A directory resource SHOULD exist at the root of the hierarchy named META-INF	Interoperation
[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.	Interoperation
[ASM12007]	Implementations of SCA MAY also raise an error if there are conflicting names exported from multiple	Development

	contributions.	
[ASM12029]	An SCA runtime MAY deploy the composites in <deployable/> elements found in the META-INF/sca-contribution.xml and META-INF/sca-contribution-generated.xml files.	Interoperation

Appendix D. Acknowledgements

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

Participants:

Participant Name	Affiliation
Bryan Aupperle	IBM
Ron Barack	SAP AG*
Michael Beisiegel	IBM
Megan Beynon	IBM
Vladislav Bezrukov	SAP AG*
Henning Blohm	SAP AG*
Fraser Bohm	IBM
David Booz	IBM
Fred Carter	AmberPoint
Martin Chapman	Oracle Corporation
Graham Charters	IBM
Shih-Chang Chen	Oracle Corporation
Chris Cheng	Primeton Technologies, Inc.
Vamsavardhana Reddy Chillakuru	IBM
Mark Combellack	Avaya, Inc.
Jean-Sebastien Delfino	IBM
David DiFranco	Oracle Corporation
Mike Edwards	IBM
Jeff Estefan	Jet Propulsion Laboratory:*
Raymond Feng	IBM
Billy Feng	Primeton Technologies, Inc.
Paul Fremantle	WSO2*
Robert Freund	Hitachi, Ltd.
Peter Furniss	Iris Financial Solutions Ltd.
Genadi Genov	SAP AG*
Mark Hapner	Sun Microsystems
Zahir HEZOUAT	IBM
Simon Holdsworth	IBM
Sabin Ielceanu	TIBCO Software Inc.
Bo Ji	Primeton Technologies, Inc.
Uday Joshi	Oracle Corporation
Mike Kaiser	IBM
Khanderao Kand	Oracle Corporation
Anish Karmarkar	Oracle Corporation
Nickolaos Kavantzas	Oracle Corporation
Rainer Kerth	SAP AG*
Dieter Koenig	IBM
Meeraj Kunnumpurath	Individual
Jean Baptiste Laviron	Axway Software*

Simon Laws	IBM
Rich Levinson	Oracle Corporation
Mark Little	Red Hat
Ashok Malhotra	Oracle Corporation
Jim Marino	Individual
Carl Mattocks	CheckMi*
Jeff Mischkinsky	Oracle Corporation
Ian Mitchell	IBM
Dale Moberg	Axway Software*
Simon Moser	IBM
Simon Nash	Individual
Peter Niblett	IBM
Duane Nickull	Adobe Systems
Eisaku Nishiyama	Hitachi, Ltd.
Sanjay Patil	SAP AG*
Plamen Pavlov	SAP AG*
Peter Peshev	SAP AG*
Gilbert Pilz	Oracle Corporation
Nilesh Rade	Deloitte Consulting LLP
Martin Raepple	SAP AG*
Luciano Resende	IBM
Michael Rowley	Active Endpoints, Inc.
Vicki Shipkowitz	SAP AG*
Ivana Trickovic	SAP AG*
Clemens Utschig - Utschig	Oracle Corporation
Scott Vorthmann	TIBCO Software Inc.
Feng Wang	Primeton Technologies, Inc.
Tim Watson	Oracle Corporation
Eric Wells	Hitachi, Ltd.
Robin Yang	Primeton Technologies, Inc.
Prasad Yendluri	Software AG, Inc.*

Appendix E. Non-Normative Text

Appendix F. Revision History

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
14 CD02-Rev2	2009-01-30	Mike Edwards	Issue 94 applied (removal of conversations)
15 CD02-Rev3	2009-01-30	Mike Edwards	Issue 98 - Section 5.3 Minor editorial cleanup (various locations) Removal of <operation/> element as decided at Jan 2009 F2F - various sections Issue 95 - Section 6.2 Issue 2 - Section 2.1 Issue 37 - Sections 2.1, 6, 12.6.1, B10 Issue 48 - Sections 5.3, A2 Issue 90 - Sections 6.1, 6.2, 6.4 Issue 64 - Sections 7, A2 Issue 100 - Section 6.2 Issue 103 - Sections 10, 12.2.2, A.13 Issue 104 - Sections 4.1.3, 5.4, 6.3 Section 3 (Quick Tour By Sample) removed by decision of Jan 2009 Assembly F2F meeting
16 CD02-Rev4	2009-02-06	Mike Edwards	All changes accepted Major Editorial work to clean out all RFC2119 wording and to ensure that no normative statements have been missed.
16 CD02-Rev6	2009-02-24	Mike Edwards	Issue 107 - sections 4, 5, 11, Appendix C Editorial updates resulting from Review Issue 34 - new section 12 inserted, + minor editorial changes in sections 4, 11 Issue 110 - Section 8.0 Issue 111 - Section 4.4, Appendix C Issue 112 - Section 4.5 Issue 113 - Section 3.3 Issue 108 - Section 13, Appendix C Minor editorial changes to the example in section 3.3
17 CD02-Rev7	2009-03-02	Mike Edwards	Editorial changes resulting from Vamsi's review of CD02 Rev6 Issue 109 - Section 8, Appendix A.2, Appendix B.3.1, Appendix C Added back @requires and @policySets to <interface/> as editorial correction since they were lost by accident in earlier revision Issue 101 - Section 13 Issue 120 - Section
18 CD02-Rev 8	2009-03-05	Mike Edwards	XSDs corrected and given new namespace. Namespace updated throughout document.
19 CD03	2009-03-05	Mike Edwards	All Changes Accepted
20 CD03	2009-03-17	Anish Karmarkar	Changed CD03 per TC's CD03/PR01 resolution. Fixed the footer, front page.
21 CD03 Rev1	2009-06-16	Mike Edwards	Issue 115 - Sections 3.1.3, 4.4, 5.3, A.2 Editorial: Use the form "portType" in all cases when referring to WSDL portType Issue 117 - Sections 4.2, 4.3, 5.0, 5.1, 5.2, 5.4, 5.4.2, 6.0, add new 7.2, old 7.2 Note: REMOVED assertions: ASM60015 ASM60015 ASM60016 ASM60017

			ASM60018 ASM60019 ASM60020 ASM60023 ASM60024 ASM80012 ASM80013 ASM80014 ASM80015 ADDED ASM70007 Issue 122 - Sections 4.3, 4.3.1, 4.3.1.1, 6.0, 8.0, 11.6 Issue 123 - Section A.2 Issue 124 - Sections A2, A5 Issue 125 - Section 7.6 Editorial - fixed broken reference links in Sections 7.0, 11.2 Issue 126 - Section 7.6 Issue 127 - Section 4.4, added Section 4.4.1 Issue 128 - Section A2 Issue 129 - Section A2 Issue 130 - multiple sections Issue 131 - Section A.11 Issue 135 - Section 8.4.2 Issue 141 - Section 4.3
22 CD03 Rev2	2009-07-28	Mike Edwards	Issue 151 - Section A.2 Issue 133 - Sections 7, 11.2 Issue 121 - Section 13.1, 13.2, C.1, C.2 Issue 134 - Section 5.2 Issue 153 - Section 3.2, 5.3.1
23 CD03 Rev3	2009-09-23	Mike Edwards	Major formatting update - all snippets and examples given a caption and consistent formatting. All references to snippets and examples updated to use the caption numbering. Issue 147 - Section 5.5.1 added Issue 136 - Section 4.3, 5.2 Issue 144 - Section 4.4 Issue 156 - Section 8 Issue 160 - Section 12.1 Issue 176 - Section A.5 Issue 180 - Section A.1 Issue 181 - Section 5.1, 5.2
24 CD03 Rev4	2009-09-23	Mike Edwards	All changes accepted Issue 157 - Section 6 removed, other changes scattered through many other sections, including the XSDs and normative statements. Issue 182 - Appendix A
25 CD03 Rev5	2009-11-20	Mike Edwards	All changes accepted Issue 138 - Section 10.3 added Issue 142 - Section 4.3 updated Issue 143 - Section 7.5 updated Issue 145 - Section 4.4 updated Issue 158 - Section 5.3.1 updated Issue 183 - Section 7.5 updated Issue 185 - Section 10.9 updated
26 CD03 Rev6	2009-12-03	Mike Edwards	All changes accepted Issue 175 - Section A2 updated Issue 177 - Section A2 updated Issue 188 - Sections 3.1.1, 3.1.2, 3.1.4, 4, 4.1, 4.2, 4.3, 5, 5.1, 5.2, 6, 6.6, 7, 7.5, 9, A2 updated Issue 192 - editorial fixes in Sections 5.1, 5.2,

			5.4.1, 5.5, 5.6.1 SCA namespace updated to http://docs.oasis-open.org/ns/opencsa/sca/200912 as decided at Dec 1st F2F meeting - changes scattered through the document Issue 137 - Sections 5.4, 7 updated Issue 189 - Section 6.5 updated
27 CD04	2009-12-09	Mike Edwards	All changes accepted
28 CD05	2010-01-12	Mike Edwards	All changes accepted Issue 215 – Section 8 and A.12
29 CD05 Rev1	2010-07-13	Bryan Aupperle	Issue 221 – Sections 3.1.3, 4.4 updated and 4.4.2 added Issue 222 – Section 8 and A.12 updated Issue 223 – Sections A.2 and A.11 updated Issue 225 – Section B.12 added Issue 228 – Section A.2 updated Issue 229 – Section 5 updated
30 CD05 Rev2	2010-08-10	Mike Edwards Bryan Aupperle	Issue 237 – Section A.1 updated Templated requirements – Section 1.4 added References to other SCA specifications updated to current drafts – Section 1.3 updated
31 CD06	2010-08-10	Mike Edwards	All changes accepted Editorial cleaning
32 WD061	2011-01-04	Mike Edwards	Issue 252 - Sections 1.2 & 12.2 updated
33 v1.2 wd02	2011-01-24	Anish Karmarkar	Applied resolutions of issues: 238, 241, 242
34 v1.2 wd03	2011-01-25	Bryan Aupperle	Syncronize v1.2 with v1.1 CSD07
35 v1.2 wd04	2011-05-24	Anish Karmarkar	Synchronize v1.2 with v1.1 WD074 (approved as csd08) Applied resolution of issue 256 Removed last paragraph of Section 7 because of resolution of issue 242 Labeled conformance items: 20101-20102, 40101-40104, 50101-50102, 60101-60109, 16001, 17001-17004
36 v1.2 wd05	2011-06-01	Anish Karmarkar	Sync v1.2 with v1.1 WD075 Fixed names of the schema includes to point to 1.2 XSDs
37 v1.2 wd06	2011-06-28	Anish Karmarkar	Issue 245 resolution and ed fixes to remove "????"