



Service Component Architecture Assembly Model Specification Version 1.1

Committee Draft 01 Revision 2 **+ Conformance**

22nd October 2008

Deleted: 23rd September,

Specification URIs:

This Version:

<http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd01.html>
<http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd01.doc>
<http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd01.pdf> (Authoritative)

Previous Version:

Latest Version:

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

Latest Approved Version:

Technical Committee:

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

Chair(s):

Martin Chapman, Oracle
Mike Edwards, IBM

Editor(s):

Michael Beisiegel, IBM
Khanderao Khand, Oracle
Anish Karmarkar, Oracle
Sanjay Patil, SAP
Michael Rowley, BEA Systems

Related work:

This specification replaces or supercedes:

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

This specification is related to:

- Service Component Architecture Policy Framework Specification Version 1.1

Declared XML Namespace(s):

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

Abstract:

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

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

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

This document describes the SCA Assembly Model, which covers

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

Status:

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

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

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

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

Notices

Copyright © OASIS® 2005, 2008. All Rights Reserved.

All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual Property Rights Policy (the "OASIS IPR Policy"). The full Policy may be found at the OASIS website.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to OASIS, except as needed for the purpose of developing any document or deliverable produced by an OASIS Technical Committee (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must be followed) or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

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

Table of Contents

1	Introduction	8
1.1	Terminology	8
1.2	Normative References	8
2	Overview	10
2.1	Diagram used to Represent SCA Artifacts	11
3	Quick Tour by Sample	13
4	Implementation and ComponentType	14
4.1	Component Type	14
4.1.1	Service	15
4.1.2	Reference	16
4.1.3	Property	18
4.1.4	Implementation	19
4.2	Example ComponentType	20
4.3	Example Implementation	20
5	Component	23
5.1	Implementation	24
5.2	Service	25
5.3	Reference	27
5.3.1	Specifying the Target Service(s) for a Reference	29
5.4	Property	30
5.5	Example Component	33
6	Composite	37
6.1	Service	39
6.1.1	Service Examples	40
6.2	Reference	41
6.2.1	Example Reference	43
6.3	Property	45
6.3.1	Property Examples	46
6.4	Wire	50
6.4.1	Wire Examples	52
6.4.2	Autowire	53
6.4.3	Autowire Examples	54
6.5	Using Composites as Component Implementations	57
6.5.1	Example of Composite used as a Component Implementation	59
6.6	Using Composites through Inclusion	60
6.6.1	Included Composite Examples	61
6.7	Composites which Include Component Implementations of Multiple Types	64
7	ConstrainingType	65
7.1	Example constrainingType	66
8	Interface	68
8.1	Local and Remotable Interfaces	69
8.2	Bidirectional Interfaces	70
8.3	Conversational Interfaces	70

Deleted: 24

Deleted: 25

Deleted: 26

Deleted: 27

Deleted: 29

Deleted: 31

Deleted: 34

Deleted: 37

Deleted: 39

Deleted: 40

Deleted: 41

Deleted: 43

Deleted: 45

Deleted: 46

Deleted: 50

Deleted: 52

Deleted: 53

Deleted: 54

Deleted: 57

Deleted: 59

Deleted: 60

Deleted: 61

Deleted: 64

Deleted: 65

Deleted: 66

Deleted: 68

Deleted: 69

Deleted: 70

Deleted: 70

Deleted: 23 September

		Deleted: 73
		Deleted: 74
8.4 SCA-Specific Aspects for WSDL Interfaces	73	Deleted: 76
Binding	74	Deleted: 76
8.5 Messages containing Data not defined in the Service Interface	76	Deleted: 76
8.6 Form of the URI of a Deployed Binding	76	Deleted: 77
8.6.1 Constructing Hierarchical URIs	76	Deleted: 77
8.6.2 Non-hierarchical URIs	77	Deleted: 78
8.6.3 Determining the URI scheme of a deployed binding	77	Deleted: 78
8.7 SCA Binding	78	Deleted: 79
8.7.1 Example SCA Binding	78	Deleted: 79
8.8 Web Service Binding	79	Deleted: 80
8.9 JMS Binding	79	Deleted: 81
9 SCA Definitions	80	Deleted: 81
10 Extension Model	81	Deleted: 83
10.1 Defining an Interface Type	81	Deleted: 84
10.2 Defining an Implementation Type	83	Deleted: 87
10.3 Defining a Binding Type	84	Deleted: 87
11 Packaging and Deployment	87	Deleted: 87
11.1 Domains	87	Deleted: 87
11.2 Contributions	87	Deleted: 88
11.2.1 SCA Artifact Resolution	88	Deleted: 89
11.2.2 SCA Contribution Metadata Document	89	Deleted: 90
11.2.3 Contribution Packaging using ZIP	90	Deleted: 91
11.3 Installed Contribution	91	Deleted: 91
11.3.1 Installed Artifact URIs	91	Deleted: 91
11.4 Operations for Contributions	91	Deleted: 92
11.4.1 install Contribution & update Contribution	92	Deleted: 92
11.4.2 add Deployment Composite & update Deployment Composite	92	Deleted: 92
11.4.3 remove Contribution	92	Deleted: 92
11.5 Use of Existing (non-SCA) Mechanisms for Resolving Artifacts	92	Deleted: 93
11.6 Domain-Level Composite	93	Deleted: 93
11.6.1 add To Domain-Level Composite	93	Deleted: 93
11.6.2 remove From Domain-Level Composite	93	Deleted: 94
11.6.3 get Domain-Level Composite	94	Deleted: 94
11.6.4 get QName Definition	94	Deleted: 95
12 Conformance	95	Deleted: 96
A. Pseudo Schema	96	Deleted: 96
A.1 ComponentType	96	Deleted: 97
A.2 Composite	97	Deleted: 99
B. XML Schemas	99	Deleted: 99
B.1 sca.xsd	99	Deleted: 99
B.2 sca-core.xsd	99	Deleted: 99
B.3 sca-binding-sca.xsd	108	Deleted: 108
B.4 sca-interface-java.xsd	109	Deleted: 109
B.5 sca-interface-wsdl.xsd	109	Deleted: 109
B.6 sca-implementation-java.xsd	110	Deleted: 110
		Deleted: 23 September

B.7 sca-implementation-composite.xsd	111	Deleted: 111
B.8 sca-definitions.xsd	111	Deleted: 111
B.9 sca-binding-webservice.xsd	112	Deleted: 112
B.10 sca-binding-jms.xsd	112	Deleted: 112
B.11 sca-policy.xsd	112	Deleted: 112
B.12 sca-contribution.xsd	112	Deleted: 112
C. SCA Concepts	114	Deleted: 114
C.1 Binding	114	Deleted: 114
C.2 Component	114	Deleted: 114
C.3 Service	114	Deleted: 114
C.3.1 Remotable Service	114	Deleted: 114
C.3.2 Local Service	115	Deleted: 115
C.4 Reference	115	Deleted: 115
C.5 Implementation	115	Deleted: 115
C.6 Interface	115	Deleted: 115
C.7 Composite	116	Deleted: 116
C.8 Composite inclusion	116	Deleted: 116
C.9 Property	116	Deleted: 116
C.10 Domain	116	Deleted: 116
C.11 Wire	116	Deleted: 116
D. Acknowledgements	117	Deleted: 117
E. Non-Normative Text	121	Deleted: 118
F. Revision History	122	Deleted: 119

1 Introduction

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

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

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

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

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

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

1.1 Terminology

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

1.2 Normative References

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

[1] SCA Java Component Implementation Specification

SCA Java Common Annotations and APIs Specification

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

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

[2] SDO Specification

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

[3] SCA Example Code document

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

[4] JAX-WS Specification

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

[5] WS-I Basic Profile

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

[6] WS-I Basic Security Profile

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

40
41 [7] Business Process Execution Language (BPEL)
42 http://www.oasis-open.org/committees/documents.php?wg_abbrev=wsbpel
43
44 [8] WSDL Specification
45 WSDL 1.1: <http://www.w3.org/TR/wsdl>
46 WSDL 2.0: <http://www.w3.org/TR/wsdl20/>
47
48 [9] SCA Web Services Binding Specification
49 http://www.osoa.org/download/attachments/35/SCA_WebServiceBindings_V100.pdf
50
51 [10] SCA Policy Framework Specification
52 http://www.osoa.org/download/attachments/35/SCA_Policy_Framework_V100.pdf
53
54 [11] SCA JMS Binding Specification
55 http://www.osoa.org/download/attachments/35/SCA_JMSBinding_V100.pdf
56
57 [12] ZIP Format Definition
58 <http://www.pkware.com/documents/casestudies/APPNOTE.TXT>
59
60 [13] Infoset Specification
61 <http://www.w3.org/TR/xml-infoset/>
62

2 Overview

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

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

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

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

Deleted: may

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

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

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

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

Deleted: may

Deleted: may

Deleted: 23 September

2.1 Diagram used to Represent SCA Artifacts

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

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

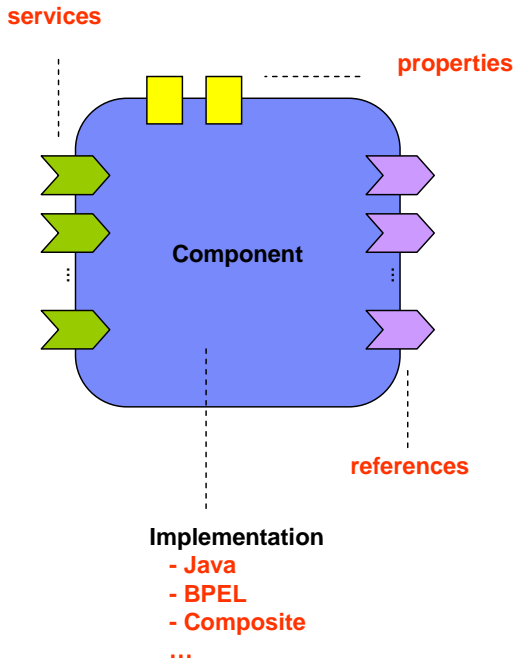


Figure 1: SCA Component Diagram

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

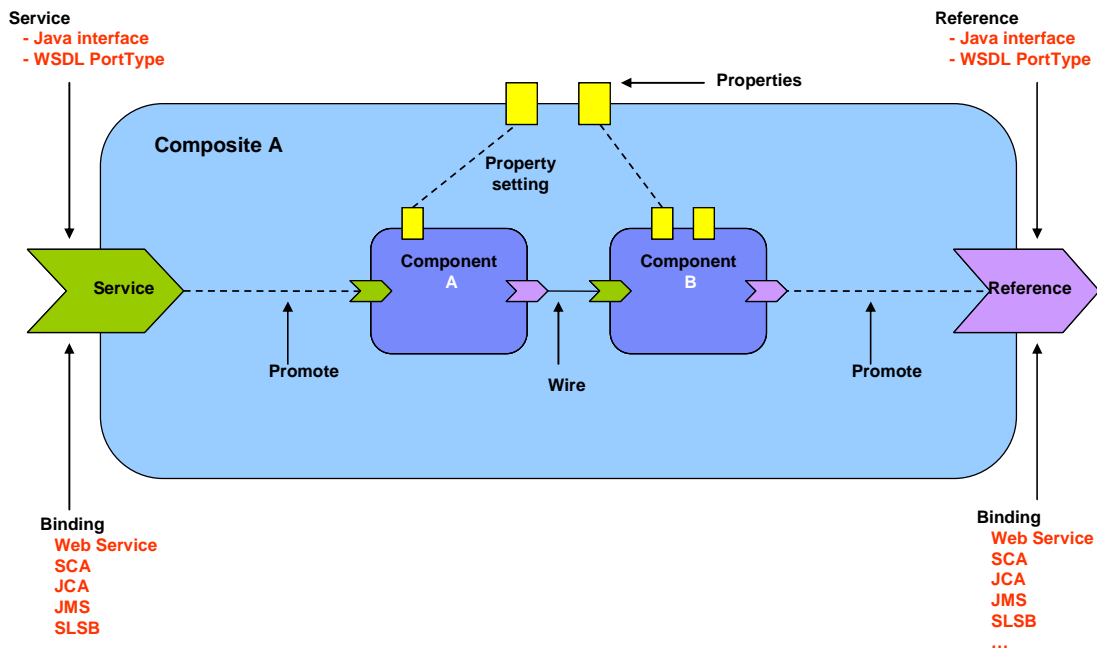


Figure 2: SCA Composite Diagram

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

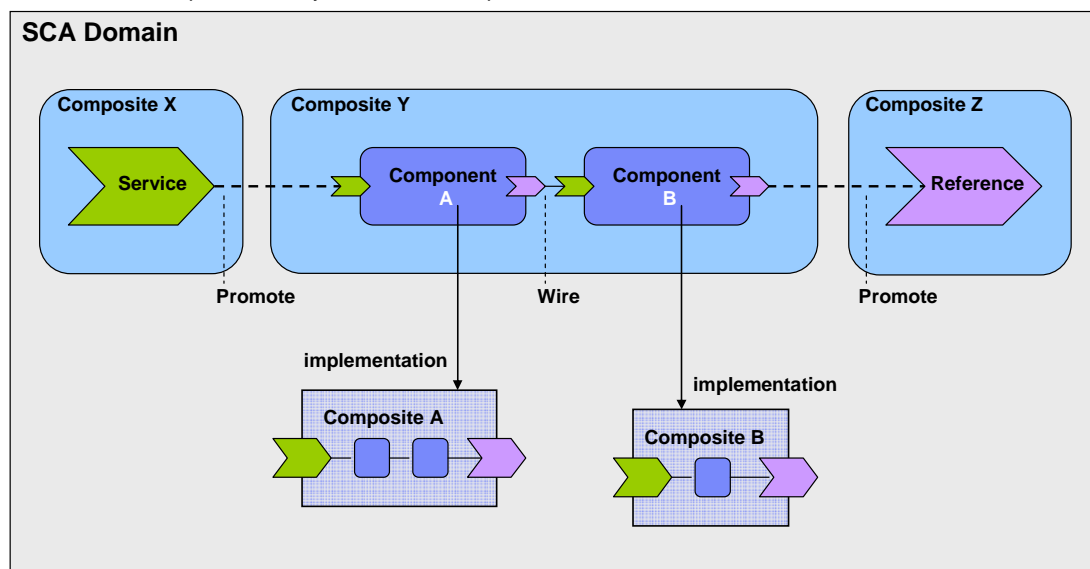


Figure 3: SCA Domain Diagram

131
132
133
134
135
136

3 Quick Tour by Sample

To be completed.

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

4 Implementation and ComponentType

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

Deleted: may

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.

Deleted: you to choose from

Deleted: may

Deleted: ay

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

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

Deleted: may be able to

Deleted: may be able

So, for example:

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

Deleted: ay

Deleted: ay

Deleted: ay

Deleted: may

Deleted: and

Deleted: ,

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

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

Deleted: may

4.1 Component Type

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

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

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

Deleted: 23 September

The component type is defined by a `componentType` element in the `componentType` file. The extension of a `componentType` file MUST be `.componentType` and its name and location depends on the type of the component implementation: the specifics are described in the respective client and implementation model specification for the implementation type.

Comment [mbgl1]: Issue 67

The following snippet shows the `componentType` schema.

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

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

</componentType>
```

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

- `constrainingType : QName (0..1)`** – If present, the `@constrainingType` attribute of a `<componentType/>` element MUST reference a `<constrainingType/>` element in the Domain through its QName. [ASM40002]. When specified, the set of services, references and properties of the implementation, plus related intents, is constrained to the set defined by the `constrainingType`. See the [ConstrainingType Section](#) for more details.

Deleted: the name of a constrainingType.

Formatted: Font color: Red

Deleted:

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

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

4.1.1 Service

A **Service** represents an addressable interface of the implementation. The service is represented by a **service element** which is a child of the `componentType` element. There can be **zero or more** service elements in a `componentType`. The following snippet shows the component type schema with the schema for a service child element:

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

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

Deleted: 23 September

```

233         <binding ... />+
234     </callback>
235 </service>
236
237 <reference ... />*
238 <property ... />*
239 <implementation ... />?
240
241 </componentType>
242

```

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

- **name : NCName (1..1)** - the name of the service. **The @name attribute of a <service/> child element of a <componentType/> MUST be unique amongst the service elements of that <componentType/>. [Error! Reference source not found.]**
- **requires : QName (0..n)** - a list of policy intents. See the [Policy Framework specification \[10\]](#) for a description of this attribute.
- **policySets : QName (0..n)** - a list of policy sets. See the [Policy Framework specification \[10\]](#) for a description of this attribute.

Formatted: Font color: Red

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

- **interface : Interface (1..1)** - A service has **one interface**, which describes the operations provided by the service. **For details on the interface element see the Interface section.**
- **binding : Binding (0..n)** - A service element has **zero or more binding elements** as children. If the binding element is not present it defaults to <binding.sca>. Details of the binding element are described in [the Bindings section](#).
- **callback (0..1) / binding : Binding (1..n)** - A service element has an optional **callback** element used if the interface has a callback defined, which has one or more **binding** elements as children. The **callback** and its binding child elements are specified if there is a need to have binding details used to handle callbacks. If the callback element is not present, the behaviour is runtime implementation dependent. **For details on callbacks, see the Bidirectional Interfaces section.**

Deleted: The interface is described by an **interface element** which is a child element of the service element.

Deleted: The binding, combined with any PolicySets in effect for the binding, must satisfy the set of policy intents for the service, as described in the [Policy Framework specification \[10\]](#).

4.1.2 Reference

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

```

273 <?xml version="1.0" encoding="ASCII"?>
274 <!-- Component type reference schema snippet -->
275 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ...
276 >
277
278     <service ... />*
279
280     <reference name="xs:NCName"
281         target="list of xs:anyURI"? autowire="xs:boolean"?
282         multiplicity="0..1 or 1..1 or 0..n or 1..n"?
283         wiredByImpl="xs:boolean"?

```

Deleted: 23 September


```

284         requires="list of xs:QName"? policySets="list of xs:QName"?>*
285     <interface ... />
286     <binding ... />*
287     <callback?
288         <binding ... />+
289     </callback>
290 </reference>
291
292 <property ... />*
293 <implementation ... />?
294
295 </componentType>
296

```

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

- **name : NCName (1..1)** - the name of the reference. The @name attribute of a <reference/> child element of a <componentType/> MUST be unique amongst the reference elements of that <componentType/>. **[Error! Reference source not found.]**
- **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".
- **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](#).
- **autowire : boolean (0..1)** - whether the reference should be autowired, as described in [the Autowire section](#). Default is false.
- **wiredByImpl : boolean (0..1)** - a boolean value, "false" by default. If set to "false", the reference is wired to the target(s) configured on the reference. If set to "true" it indicates that the target of the reference is set at runtime by the implementation code (eg by the code obtaining an endpoint reference by some means and setting this as the target of the reference through the use of programming interfaces defined by the relevant Client and Implementation specification). If @wiredByImpl is set to "true", then any reference targets configured for this reference MUST be ignored by the runtime. **[ASM40006] It is recommended that any references with @wiredByImpl = "true" are left unwired.**
- **requires : QName (0..n)** - a list of policy intents. See the [Policy Framework specification \[10\]](#) for a description of this attribute.
- **policySets : QName (0..n)** - a list of policy sets. See the [Policy Framework specification \[10\]](#) for a description of this attribute.

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

- **interface : Interface (1..1)** - A reference has **one interface**, which describes the operations required by the reference. The interface is described by an **interface element** which is a child element of the reference element. For details on the interface element see [the Interface section](#).
- **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](#).

Formatted: Font color: Red

Deleted: , which indicates that the implementation wires this reference dynamically.

Deleted: If "true" is set, then the reference

Formatted: Font color: Red

Deleted: should

Deleted: not be wired statically within a composite, but

Deleted: The binding, combined with any PolicySets in effect for the binding, must satisfy the set of policy intents for the reference, as described in [the Policy Framework specification \[10\]](#).

Deleted: 23 September

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

- **callback (0..1) / binding : Binding (1..n)** - A **reference** element has an optional **callback** element used if the interface has a callback defined, which has one or more **binding** elements as children. The **callback** and its binding child elements are specified if there is a need to have binding details used to handle callbacks. If the callback element is not present, the behaviour is runtime implementation dependent. [For details on callbacks, see the Bidirectional Interfaces section.](#)

4.1.3 Property

Properties allow for the configuration of an implementation with externally set values. Each Property is defined as a property element. The componentType element can have zero or more property elements as its children. The following snippet shows the component type schema with the schema for a reference child element:

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

  <service ... />*
  <reference ... >*

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

  <implementation ... />?
</componentType>
```

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

- **name : NCName (1..1)** - the name of the property. [The @name attribute of a <property/> child element of a <componentType/> MUST be unique amongst the property elements of that <componentType/>. \[ASM40005\]](#)
- one of **(1..1)**:
 - **type : QName** - the type of the property defined as the qualified name of an XML schema type. [The value of the property @type attribute MUST be the QName of an XML schema type. \[ASM40007\]](#)
 - **element : QName** - the type of the property defined as the qualified name of an XML schema global element – the type is the type of the global element. [The value of the property @element attribute MUST be the QName of an XSD global element. \[ASM40008\]](#)

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Deleted: 23 September

- **many : boolean (0..1)** - (optional) whether the property is single-valued (false) or multi-valued (true). In the case of a multi-valued property, it is presented to the implementation as a collection of property values.
- **mustSupply : boolean (0..1)** - whether the property value must be supplied by the component that uses the implementation – when mustSupply="true" the component must supply a value since the implementation has no default value for the property. A default-property-value should only be supplied when mustSupply="false" (the default setting for the mustSupply attribute), since the implication of a default value is that it is used only when a value is not supplied by the using component.
- **file : anyURI (0..1)** - a dereferencable URI to a file containing a value for the `property`.

Comment [mbgl2]: Issue 68

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

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

Comment [mbgl3]: Issue 38

Deleted: MAY

Deleted: MAY

Deleted: MUST

Deleted: and the SCA runtime MUST ensure that any implementation default value is replaced

Formatted: Font color: Red

Deleted: .

4.1.4 Implementation

Implementation represents characteristics inherent to the implementation itself, in particular intents and policies. See the [Policy Framework specification \[10\]](#) for a description of intents and policies. The following snippet shows the component type schema with the schema for a implementation child element:

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

  <service ... />*
  <reference ... >*
  <property ... />*

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

</componentType>
```

Formatted: Space Before: 0 pt

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

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

Deleted: 23 September

4.2 Example ComponentType

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

```
<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">

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

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

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

</componentType>
```

4.3 Example Implementation

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

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

```
package services.account;

@Remotable
public interface AccountService {

    AccountReport getAccountReport(String customerID);
}
```

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

```
package services.account;

import java.util.List;

import commonj.sdo.DataFactory;
```

```

484     import org.osoa.sca.annotations.Property;
485     import org.osoa.sca.annotations.Reference;
486
487     import services.accountdata.AccountDataService;
488     import services.accountdata.CheckingAccount;
489     import services.accountdata.SavingsAccount;
490     import services.accountdata.StockAccount;
491     import services.stockquote.StockQuoteService;
492
493     public class AccountServiceImpl implements AccountService {
494
495         @Property
496         private String currency = "USD";
497
498         @Reference
499         private AccountDataService accountDataService;
500         @Reference
501         private StockQuoteService stockQuoteService;
502
503         public AccountReport getAccountReport(String customerID) {
504
505             DataFactory dataFactory = DataFactory.INSTANCE;
506             AccountReport accountReport = (AccountReport)dataFactory.create(AccountReport.class);
507             List accountSummaries = accountReport.getAccountSummaries();
508
509             CheckingAccount checkingAccount = accountDataService.getCheckingAccount(customerID);
510             AccountSummary checkingAccountSummary =
511 (AccountSummary)dataFactory.create(AccountSummary.class);
512             checkingAccountSummary.setAccountNumber(checkingAccount.getAccountNumber());
513             checkingAccountSummary.setAccountType("checking");
514             checkingAccountSummary.setBalance(fromUSDollarToCurrency(checkingAccount.getBalance()));
515             accountSummaries.add(checkingAccountSummary);
516
517             SavingsAccount savingsAccount = accountDataService.getSavingsAccount(customerID);
518             AccountSummary savingsAccountSummary =
519 (AccountSummary)dataFactory.create(AccountSummary.class);
520             savingsAccountSummary.setAccountNumber(savingsAccount.getAccountNumber());
521             savingsAccountSummary.setAccountType("savings");
522             savingsAccountSummary.setBalance(fromUSDollarToCurrency(savingsAccount.getBalance()));
523             accountSummaries.add(savingsAccountSummary);
524
525             StockAccount stockAccount = accountDataService.getStockAccount(customerID);
526             AccountSummary stockAccountSummary =
527 (AccountSummary)dataFactory.create(AccountSummary.class);
528             stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());
529             stockAccountSummary.setAccountType("stock");
530             float balance=
531 (stockQuoteService.getQuote(stockAccount.getSymbol()))*stockAccount.getQuantity();
532             stockAccountSummary.setBalance(fromUSDollarToCurrency(balance));
533             accountSummaries.add(stockAccountSummary);
534
535             return accountReport;

```

```

536     }
537
538     private float fromUSDollarToCurrency(float value){
539
540         if (currency.equals("USD")) return value; else
541         if (currency.equals("EURO")) return value * 0.8f; else
542         return 0.0f;
543     }
544 }

```

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

```

549 <?xml version="1.0" encoding="ASCII"?>
550 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
551               xmlns:xsd="http://www.w3.org/2001/XMLSchema">
552
553     <service name="AccountService">
554         <interface.java interface="services.account.AccountService"/>
555     </service>
556     <reference name="accountDataService">
557         <interface.java
558 interface="services.accountdata.AccountDataService"/>
559     </reference>
560     <reference name="stockQuoteService">
561         <interface.java
562 interface="services.stockquote.StockQuoteService"/>
563     </reference>
564
565     <property name="currency" type="xsd:string">USD</property>
566
567 </componentType>

```

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

Comment [ME4]: Conformance work reached here

5 Component

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

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

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

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Component schema snippet -->
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/scs/200712" ... >
  ...
  <component name="xs:NCName" autowire="xs:boolean"?
    requires="list of xs:QName"? policySets="list of xs:QName"?
    constrainingType="xs:QName"?>*
    <implementation ... />?
    <service ... />*
    <reference ... />*
    <property ... />*
  </component>
  ...
</composite>
```

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

- name : NCName (1..1)** – the name of the component. The @name attribute of a <service/> child element of a <componentType/> MUST be unique amongst the service elements of that <componentType/>. [ASM50001]
- autowire : boolean (0..1)** – whether contained component references should be autowired, as described in [the Autowire section](#). Default is false.
- requires : QName (0..n)** – a list of policy intents. See the [Policy Framework specification \[10\]](#) for a description of this attribute.
- policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification \[10\]](#) for a description of this attribute.
- constrainingType : QName (0..1)** – the name of a constrainingType. When specified, the set of services, references and properties of the component, plus related intents, is constrained to the set defined by the constrainingType. See [the ConstrainingType Section](#) for more details.

Deleted: The name must be unique across all the components in the composite.

Formatted: Font color: Red

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

- implementation : ComponentImplementation (0..1)** – [see component implementation section](#).

Deleted: 23 September

- **service** : *ComponentService* (0..n) – [see component service section.](#)
- **reference** : *ComponentReference* (0..n) – [see component reference section.](#)
- **property** : *ComponentProperty* (0..n) – [see component property section.](#)

5.1 Implementation

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

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

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

For example the elements **implementation.java**, **implementation.bpel**, **implementation.cpp**, and **implementation.c** point to Java, BPEL, C++, and C implementation types respectively. **implementation.composite** points to the use of an SCA composite as an implementation. **implementation.spring** and **implementation.ejb** are used for Java components written to the Spring framework and the Java EE EJB technology respectively.

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

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

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

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

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

Comment [mbgl5]: Issue 69 part 1

Deleted: 23 September

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

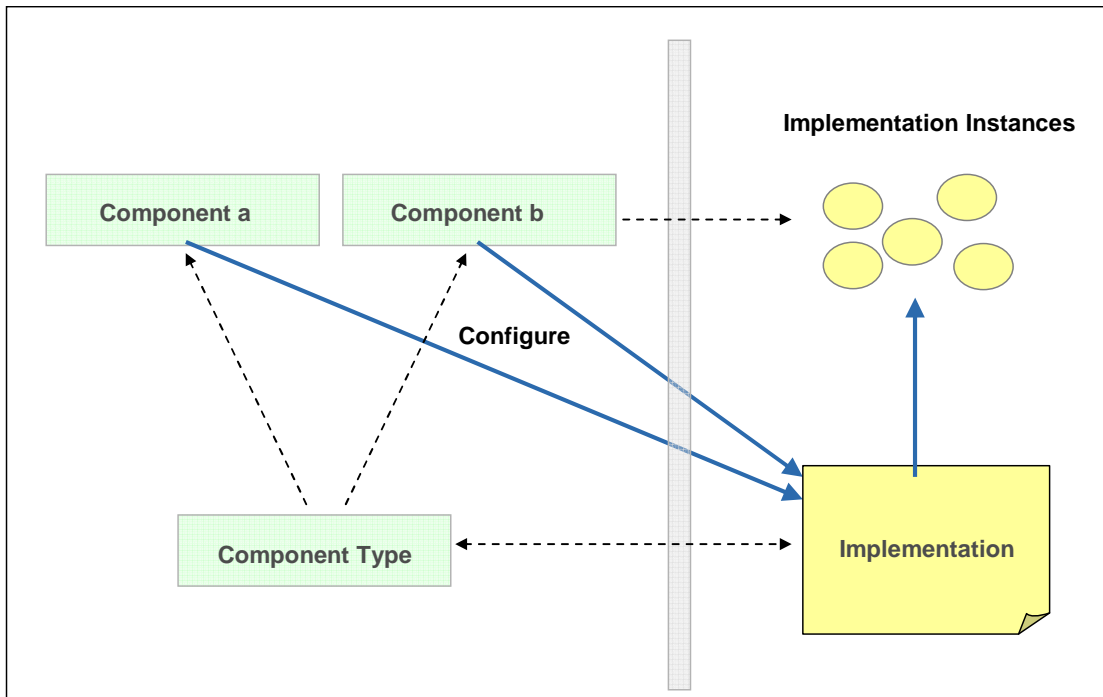


Figure 4: Relationship of Component and Implementation

5.2 Service

The component element can have **zero or more service elements** as children which are used to configure the services of the component. The services that can be configured are defined by the implementation. The following snippet shows the component schema with the schema for a service child element:

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

```

684         <callback>?
685             <binding ... />+
686         </callback>
687     </service>
688     <reference ... />*
689     <property ... />*
690 </component>
691 ...
692 </composite>
693

```

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

- **name : NCName (1..1)** - the name of the service. The @name attribute of a service element of a <component/> MUST be unique amongst the service elements of that <component/> [ASM50002]. The @name attribute of a service element of a <component/> MUST match the @name attribute of a service element of the componentType of the <implementation/> child element of the component. [ASM50003]
- **requires : QName (0..n)** - a list of policy intents. See the [Policy Framework specification \[10\]](#) for a description of this attribute.
Note: The effective set of policy intents for the service consists of any intents explicitly stated in this requires attribute, combined with any intents specified for the service by the implementation.
- **policySets : QName (0..n)** - a list of policy sets. See the [Policy Framework specification \[10\]](#) for a description of this attribute.

Deleted: Has to match a name of a service defined by the implementation.

Formatted: Font color: Red

Formatted: Font color: Red

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

- **interface : Interface (0..1)** - A service has **zero or one interface**, which describes the operations provided by the service. The interface is described by an **interface element** which is a child element of the service element. If no interface is specified, then the interface specified for the service in the componentType of the implementation is in effect. If a <service/> element has an interface subelement specified, the interface MUST provide a compatible subset of the interface declared on the componentType of the implementation. [ASM50004]. For details on the interface element see the Interface section.
- **binding : Binding (0..n)** - A service element has **zero or more binding elements** as children. If no binding elements are specified for the service, then the bindings specified for the equivalent service in the componentType of the implementation MUST be used, but if the componentType also has no bindings specified, then <binding.sca/> MUST be used as the binding. If binding elements are specified for the service, then those bindings MUST be used and they override any bindings specified for the equivalent service in the componentType of the implementation. [ASM50005]. Details of the binding element are described in the [Bindings section](#). The binding, combined with any PolicySets in effect for the binding, needs to satisfy the set of policy intents for the service, as described in the [Policy Framework specification \[10\]](#).
- **callback (0..1) / binding : Binding (1..n)** - A service element has an optional **callback** element used if the interface has a callback defined, which has one or more **binding** elements as children. The **callback** and its binding child elements are specified if there is a need to have binding details used to handle callbacks. If the callback element is present and contains one or more binding child elements, then those bindings MUST be used for the callback. [ASM50006]. If the callback element is not present, the behaviour is runtime implementation dependent.

Deleted: by

Deleted: If an interface is specified it must provide a compatible subset of the interface provided by the implementation, i.e. provide a subset of the operations defined by the implementation for the service.

Formatted: Font color: Red

Deleted: If no bindings are specified, then the bindings specified for the service by the implementation are in effect. If bindings are specified, then those bindings override the bindings specified by the implementation.

Formatted: Font color: Red

Deleted: must

Formatted: Font color: Red

Deleted: 23 September

5.3 Reference

The component element can have **zero or more reference elements** as children which are used to configure the references of the component. The references that can be configured are defined by the implementation. The following snippet shows the component schema with the schema for a reference child element:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Component Reference schema snippet -->
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
  ...
  <component ... >*
    <implementation ... />?
    <service ... />*
    <reference name="xs:NCName"
      target="list of xs:anyURI"? autowire="xs:boolean"?
      multiplicity="0..1 or 1..1 or 0..n or 1..n"?
      wiredByImpl="xs:boolean"? requires="list of xs:QName"?
      policySets="list of xs:QName"?>*
    <interface ... />?
    <binding uri="xs:anyURI"? requires="list of xs:QName"?
      policySets="list of xs:QName"?/>*
    <callback?
      <binding ... />+
    </callback>
  </reference>
  <property ... />*
</component>
  ...
</composite>
```

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

- name : NCName (1..1)** – the name of the reference. The @name attribute of a service element of a <component/> MUST be unique amongst the service elements of that <component/> [ASM50007] The @name attribute of a reference element of a <component/> MUST match the @name attribute of a reference element of the componentType of the <implementation/> child element of the component. [ASM50008]
- autowire : boolean (0..1)** – whether the reference should be autowired, as described in the **Autowire section**. Default is false.
- requires : QName (0..n)** – a list of policy intents. See the **Policy Framework specification [10]** for a description of this attribute.
Note: The effective set of policy intents for the reference consists of any intents explicitly stated in this requires attribute, combined with any intents specified for the reference by the implementation.
- policySets : QName (0..n)** – a list of policy sets. See the **Policy Framework specification [10]** for a description of this attribute.

Deleted: Has to match a name of a reference defined by the implementation.

Formatted: Font color: Red

Formatted: Font color: Red

Deleted: 23 September

- **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

Deleted: on

Deleted: The value can only be equal or further restrict, i.e. 0..n to 0..1 or 1..n to 1..1.

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]

Formatted: Indent: Left: 1.9 cm

Formatted: Font color: Red

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

- **target : anyURI (0..n)** – a list of one or more of target service URI's, depending on multiplicity setting. Each value wires the reference to a component service that resolves the reference. For more details on wiring see [the section on Wires](#). Overrides any target specified for this reference on the implementation.
- **wiredByImpl : boolean (0..1)** – a boolean value, "false" by default, which indicates that the implementation wires this reference dynamically. If set to "true" it indicates that the target of the reference is set at runtime by the implementation code (eg by the code obtaining an endpoint reference by some means and setting this as the target of the reference through the use of programming interfaces defined by the relevant Client and Implementation specification). If @wiredByImpl="true" is set for a reference, then the reference MUST NOT be wired statically within a composite, but left unwired. [ASM50010]

Deleted: If "true" is set, then the reference should not be wired statically within a composite, but left unwired.

Formatted: Font color: Red

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

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

Deleted: by

Deleted: If an interface is specified it must provide a compatible superset of the interface provided by the implementation, i.e. provide a superset of the operations defined by the implementation for the reference.

Formatted: Font color: Red

Deleted: If no bindings are specified, then the bindings specified for the reference by the implementation are in effect. If any bindings are specified, then those bindings override any and all the bindings specified by the implementation.

Formatted: Font color: Red

Deleted: must

Deleted: 23 September

A reference identifies zero or more target services that satisfy the reference. This can be done in a number of ways, which are fully described in section "5.3.1 Specifying the Target Service(s) for a Reference"

- **callback (0..1) / binding : Binding (1..n)** - A **reference** element has an optional **callback** element used if the interface has a callback defined, which has one or more **binding** elements as children. The **callback** and its binding child elements are specified if

there is a need to have binding details used to handle callbacks. If the callback element is present and contains one or more binding child elements, then those bindings MUST be used for the callback. [ASM50006] If the callback element is not present, the behaviour is runtime implementation dependent.

Comment [ME6]: Note that this is a repeated conformance point - the same applies to callback elements under services.

5.3.1 Specifying the Target Service(s) for a Reference

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

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

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, the child binding elements of that reference MUST NOT identify target services using the @uri attribute or using binding specific attributes or elements. [ASM50026]
- If a binding element has a value specified for a target service using its @uri attribute, the binding element MUST NOT identify target services using binding specific attributes or elements. [ASM50015]
- It is possible that a particular binding type MAY require that the address of a target service uses more than a simple URI. In such cases, the @uri attribute MUST NOT be used to identify the target service - instead, binding specific attributes and/or child elements must be used. [ASM50016]
- When the reference has a value specified in its @target attribute, one of the child binding elements MUST be used on each wire created by the @target attribute, or the sca binding, if no binding is specified. [ASM50017]

Deleted: If @wiredByImpl="true", other methods of specifying the target service MUST NOT be used.

Formatted: Font color: Red

Deleted: 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.

Formatted: Font color: Red

Deleted: If a reference has a value specified for one or more target services in its @target attribute, the child binding elements of that reference MUST NOT identify target services using the @uri attribute...

Formatted: Font color: Red

Deleted: If a binding element has a value ...

Formatted: Font color: Red

Deleted: It is possible that a particular binding type...

Formatted: Font color: Red

Deleted: When the reference has a value ...

Formatted: Font color: Red

Deleted: A reference with multiplicity 0..1 or 0..

Formatted: Font color: Red

Deleted: A reference with multiplicity 0..1 or 1..

Formatted: Font color: Red

Deleted: A reference with multiplicity 1..1 or 1..

Formatted: Font color: Red

Deleted: A reference with multiplicity 0..n or 1..

Formatted: Font color: Red

Deleted: 23 September

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

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

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

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

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

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

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

Deleted: Where it is detected that the above rules have been violated, either at deployment or at execution time, an SCA Runtime MUST generate an error no later than when the reference is invoked by the component implementation.

Formatted: Font color: Red

Deleted: Some errors can be detected at deployment time. 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. In these cases, an error SHOULD be generated by the SCA runtime at deployment time.

Formatted: Font color: Red

Deleted: Other errors can only be checked at runtime.

Deleted: In these cases, the SCA runtime MUST generate an error no later than when the reference is invoked by the component implementation.¶

Deleted: MUST

Deleted: 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 described in this section.

Formatted: Font color: Red

Deleted: 23 September

5.4 Property

The component element has **zero or more property elements** as its children, which are used to configure data values of properties of the implementation. Each property element provides a value for the named property, which is passed to the implementation. The properties that can be configured and their types are defined by the 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. This can be used only when the type of the property is specified using a XML Schema simple type and a single value is specified.

For example,

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

- As a value, supplied as the content of the **value** element(s) children of the property element. This can be used only when the type of the property is specified using a XML Schema simple type or a XML Schema complex type.

For example,

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

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

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

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

- 927 <value>USDollar</value>
 928 </property>
- 929 • property defined using a XML Schema complex type and which contains a single
 930 value
 931 <property name="complexFoo">
 932 <value attr="bar">
 933 <foo:a>TheValue</foo:a>
 934 <foo:b>InterestingURI</foo:b>
 935 </value>
 936 </property>
 - 937 • property defined using a XML Schema complex type and which contains multiple
 938 values
 939 <property name="complexBar">
 940 <value anotherAttr="foo">
 941 <bar:a>AValue</bar:a>
 942 <bar:b>InterestingURI</bar:b>
 943 </value>
 944 <value attr="zing">
 945 <bar:a>BValue</bar:a>
 946 <bar:b>BoringURI</bar:b>
 947 </value>
 948 </property>
 - 949 • As a value, supplied as the content of the property element.
 950 This can be used only when the type of the property is specified using a XML Schema
 951 global element declaration.
 952 For example,
 953 • property defined using a XML Schema global element declaration and which
 954 contains a single value
 955 <property name="foo">
 956 <foo:SomeGED ...>...</foo:SomeGED>
 957 </property>
 - 958 • property defined using a XML Schema global element declaration and which
 959 contains multiple values
 960 <property name="bar">
 961 <bar:SomeOtherGED ...>...</bar:SomeOtherGED>
 962 <bar:SomeOtherGED ...>...</bar:SomeOtherGED>
 963 </property>
 - 964 • By referencing a Property value of the composite which contains the component. The
 965 reference is made using the **source** attribute of the property element.
 966 The form of the value of the source attribute follows the form of an XPath expression.
 967 This form allows a specific property of the composite to be addressed by name. Where the
 968 property is complex, the XPath expression can be extended to refer to a sub-part of the
 969 complex value.
 970 The form of the value of the source attribute follows the form of an XPath expression.
 971 This form allows a specific property of the composite to be addressed by name. Where the
 972 property is complex, the XPath expression can be extended to refer to a sub-part of the
 973 complex value.

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

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

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

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

When a property has multiple values set, they MUST all be contained within the same property element. Two sibling property elements with the same value for the @name attribute is an error.

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

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

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

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

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

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

- **name : NCName (1..1)** – the name of the property. Has to match a name of a property defined by the implementation
- zero or one of **(0..1)**:

- 1017 ○ **type : QName** – the type of the property defined as the qualified name of an XML
1018 schema type
- 1019 ○ **element : QName** – the type of the property defined as the qualified name of an
1020 XML schema global element – the type is the type of the global element
- 1021 ▪ **source : string (0..1)** – an XPath expression pointing to a property of the containing
1022 composite from which the value of this component property is obtained.
- 1023 ▪ **file : anyURI (0..1)** – a dereferencable URI to a file containing a value for the property
- 1024 ▪ **many : boolean (0..1)** – (optional) whether the property is single-valued (false) or
1025 multi-valued (true). Overrides the many specified for this property on the implementation.
1026 The value can only be equal or further restrict, i.e. if the implementation specifies many
1027 true, then the component can say false. In the case of a multi-valued property, it is
1028 presented to the implementation as a Collection of property values.
- 1029 ▪ **mustSupply : boolean (0..1)** - whether the property value must be supplied by the
1030 component – when mustSupply="true" the component must supply a value since the
1031 implementation has no default value for the property.
- 1032 ▪ **value : string (0..1)** - the value of the property if the property is defined using a simple
1033 type. This property cannot be used if multiple values are specified (for multivalued
1034 properties).

1035 The **component property** element has the following **child element**:

1036 **value :any (0..n)** - A property has **zero or more**, value elements that specify the value(s) of a
1037 property that is defined using a XML Schema type. If the property is single-valued, this element
1038 MUST NOT occur more than once. This element MUST NOT be used when the @value attribute is
1039 used to specify the property value.

1040 5.5 Example Component

1041

1042 The following figure shows the **component symbol** that is used to represent a component in an
1043 assembly diagram.

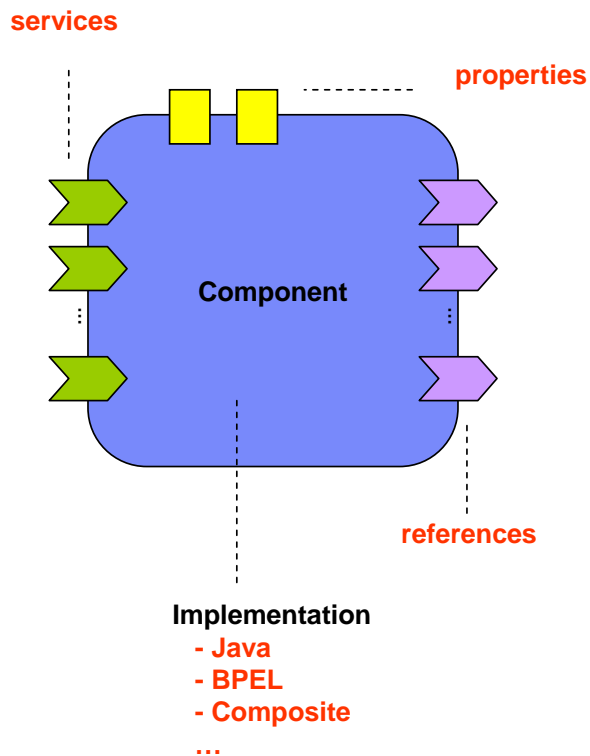


Figure 5: Component symbol

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

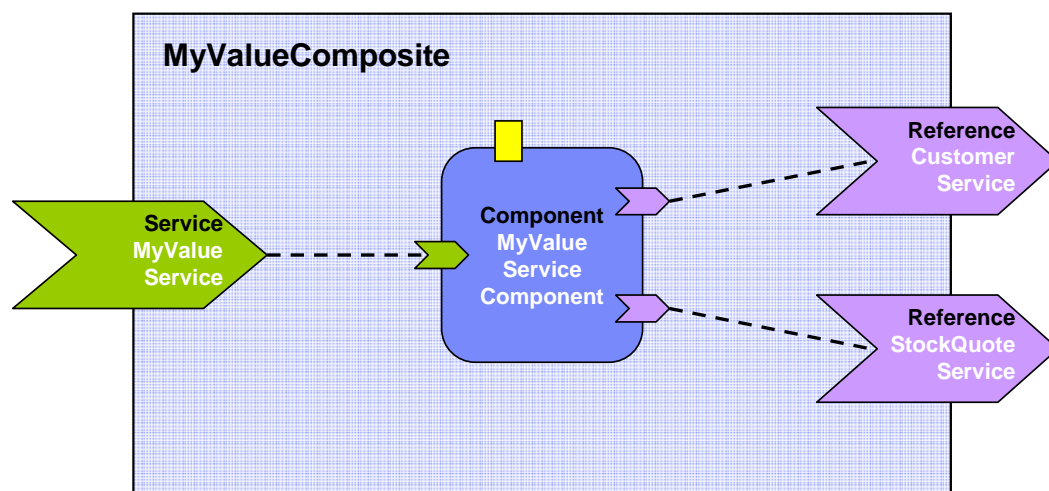


Figure 6: Assembly diagram for MyValueComposite

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

```
<?xml version="1.0" encoding="ASCII"?>
<!-- MyValueComposite_1 example -->
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
            targetNamespace="http://foo.com"
            name="MyValueComposite" >

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

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

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

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

</composite>
```

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

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

```
<?xml version="1.0" encoding="ASCII"?>
<!-- MyValueComposite_2 example -->
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
            targetNamespace="http://foo.com"
            name="MyValueComposite" >

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

```

1097     <component name="MyValueServiceComponent">
1098         <implementation.java
1099 class="services.myvalue.MyValueServiceImpl"/>
1100         <property name="currency">EURO</property>
1101         <property name="currency">Yen</property>
1102         <property name="currency">USDollar</property>
1103         <reference name="customerService"
1104             target="InternalCustomer/customerService"/>
1105         <reference name="StockQuoteService"/>
1106     </component>
1107
1108     ...
1109
1110     <reference name="CustomerService"
1111         promote="MyValueServiceComponent/customerService"/>
1112
1113     <reference name="StockQuoteService"
1114         promote="MyValueServiceComponent/StockQuoteService"/>
1115
1116 </composite>

```

1117this assumes that the composite has another component called InternalCustomer (not shown)
1118 which has a service to which the customerService reference of the MyValueServiceComponent is
1119 wired as well as being promoted externally through the composite reference CustomerService.
1120

6 Composite

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

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

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

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

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

```
<?xml version="1.0" encoding="ASCII"?>
<!-- Composite schema snippet -->
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
    targetNamespace="xs:anyURI"
    name="xs:NCName" local="xs:boolean"?
    autowire="xs:boolean"? constrainingType="QName"?
    requires="list of xs:QName"? policySets="list of xs:QName"?>

    <include ... />*

    <service ... />*
    <reference ... />*
    <property ... />*

    <component ... />*

    <wire ... />*

</composite>
```

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

- **name : NCName (1..1)** – the name of the composite. The form of a composite name is an XML QName, in the namespace identified by the targetNamespace attribute.
- **targetNamespace : anyURI (0..1)** – an identifier for a target namespace into which the composite is declared
- **local : boolean (0..1)** – whether all the components within the composite must all run in the same operating system process. local="true" means that all the components must run in the same process. local="false", which is the default, means that different components within the composite may run in different operating system processes and they may even run on different nodes on a network.
- **autowire : boolean (0..1)** – whether contained component references should be autowired, as described in [the Autowire section](#). Default is false.
- **constrainingType : QName (0..1)** – the name of a constrainingType. When specified, the set of services, references and properties of the composite, plus related intents, is constrained to the set defined by the constrainingType. See [the ConstrainingType Section](#) for more details.
- **requires : QName (0..n)** – a list of policy intents. See the [Policy Framework specification \[10\]](#) for a description of this attribute.
- **policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification \[10\]](#) for a description of this attribute.

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

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

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

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

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

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

6.1 Service

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

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

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

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

- **name : NCName (1..1)** – the name of the service, the name MUST be unique across all the composite services in the composite. The name of the composite service can be different from the name of the promoted component service.
- **promote : anyURI (1..1)** – identifies the promoted service, the value is of the form <component-name>/<service-name>. The service name is optional if the target component only has one service. The same component service can be promoted by more than one composite service.
- **requires : QName (0..n)** – a list of required policy intents. See the [Policy Framework specification \[10\]](#) for a description of this attribute. Specified **required intents** add to or further qualify the required intents defined by the promoted component service.
- **policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification \[10\]](#) for a description of this attribute.

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

- **interface : Interface (0..1)** – If an **interface** is specified it must be the same or a compatible subset of the interface provided by the promoted component service, i.e. provide a subset of the operations defined by the component service. The interface is

- described by **zero or one interface element** which is a child element of the service element. For details on the interface element see [the Interface section](#).
- **binding : Binding (0..n)** - If bindings are specified they **override** the bindings defined for the promoted component service from the composite service perspective. The bindings defined on the component service are still in effect for local wires within the composite that target the component service. A service element has zero or more **binding elements** as children. Details of the binding element are described in the [Bindings section](#). For more details on wiring see [the Wiring section](#).
 - **callback (0..1) / binding : Binding (1..n)** - A service element has an optional **callback** element used if the interface has a callback defined,, which has one or more **binding** elements as children. The **callback** and its binding child elements are specified if there is a need to have binding details used to handle callbacks. If the callback element is not present, the behaviour is runtime implementation dependent.

6.1.1 Service Examples

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

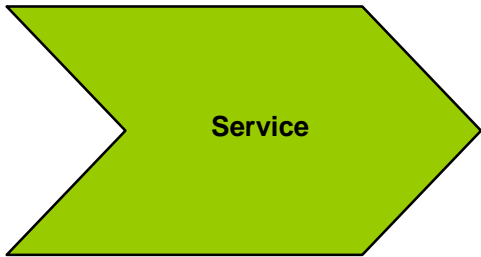


Figure 7: Service symbol

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

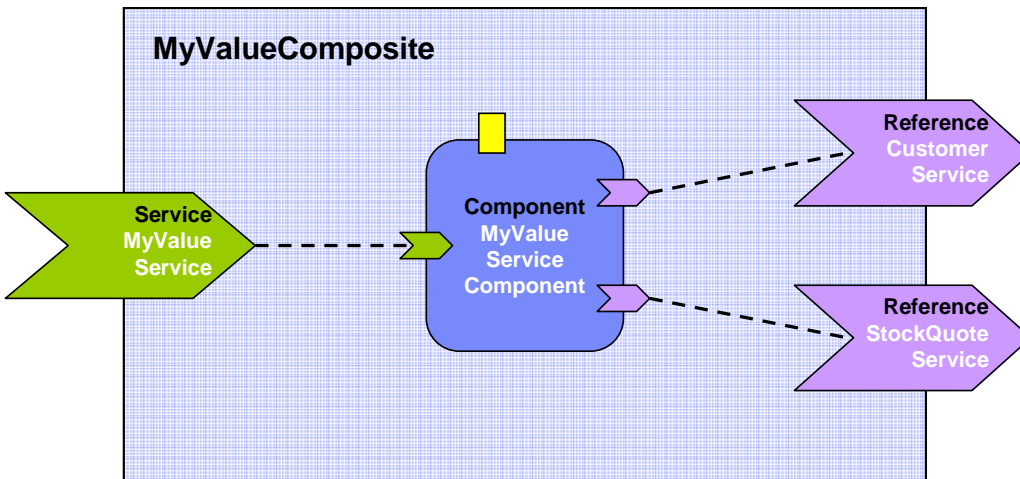


Figure 8: MyValueComposite showing Service

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

```
<?xml version="1.0" encoding="ASCII"?>
<!-- MyValueComposite_4 example -->
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
  targetNamespace="http://foo.com"
  name="MyValueComposite" >
  ...

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

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

6.2 Reference

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

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

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

```

1328 <!-- Composite Reference schema snippet -->
1329 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
1330   ...
1331   <reference name="xs:NCName" target="list of xs:anyURI"?
1332     promote="list of xs:anyURI" wiredByImpl="xs:boolean"?
1333     multiplicity="0..1 or 1..1 or 0..n or 1..n"?
1334     requires="list of xs:QName"? policySets="list of xs:QName"?>*
1335   <interface ... />?
1336   <binding ... />*
1337   <callback?
1338     <binding ... />+
1339   </callback>
1340 </reference>
1341   ...
1342 </composite>

```

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

- **name : NCName (1..1)** – the name of the reference. The name must be unique across all the composite references in the composite. The name of the composite reference can be different then the name of the promoted component reference.
- **promote : anyURI (1..n)** – identifies one or more promoted component references. The value is a list of values of the form <component-name>/<reference-name> separated by spaces. The specification of the reference name is optional if the component has only one reference.

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

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

- the interfaces of the component references are the same, or if the composite reference itself declares an interface then all the component references must have interfaces which are compatible with the composite reference interface
- the multiplicities of the component references are compatible, i.e one can be the restricted form of the another, which also means that the composite reference carries the restricted form either implicitly or explicitly
- the intents declared on the component references must be compatible – the intents which apply to the composite reference in this case are the union of the required intents specified for each of the promoted component references. If any intents contradict (eg mutually incompatible qualifiers for a particular intent) then there is an error.
- **requires : QName (0..n)** – a list of required policy intents. See the [Policy Framework specification \[10\]](#) for a description of this attribute. Specified **required intents** add to or further qualify the required intents defined for the promoted component reference.
- **policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification \[10\]](#) for a description of this attribute.
- **multiplicity : 0..1|1..1|0..n|1..n (1..1)** – Defines the number of wires that can connect the reference to target services. The multiplicity can have the following values

Deleted: 23 September

- 1376 ○ 0..1 – zero or one wire can have the reference as a source
- 1377 ○ 1..1 – one wire can have the reference as a source
- 1378 ○ 0..n - zero or more wires can have the reference as a source
- 1379 ○ 1..n – one or more wires can have the reference as a source
- 1380 The value specified for the **multiplicity** attribute has to be compatible with the multiplicity
- 1381 specified on the component reference, i.e. it has to be equal or further restrict. So a
- 1382 composite reference of multiplicity 0..1 or 1..1 can be used where the promoted
- 1383 component reference has multiplicity 0..n and 1..n respectively. However, a composite
- 1384 reference of multiplicity 0..n or 1..n cannot be used to promote a component reference of
- 1385 multiplicity 0..1 or 1..1 respectively.
- 1386 • **target : anyURI (0..n)** – a list of one or more of target service URI's, depending on
- 1387 multiplicity setting. Each value wires the reference to a service in a composite that uses
- 1388 the composite containing the reference as an implementation for one of its components. For
- 1389 more details on wiring see [the section on Wires](#).
- 1390 • **wiredByImpl : boolean (0..1)** – a boolean value, "false" by default, which indicates that
- 1391 the implementation wires this reference dynamically. If set to "true" it indicates that the
- 1392 target of the reference is set at runtime by the implementation code (eg by the code
- 1393 obtaining an endpoint reference by some means and setting this as the target of the
- 1394 reference through the use of programming interfaces defined by the relevant Client and
- 1395 Implementation specification). If "true" is set, then the reference should not be wired
- 1396 statically within a using composite, but left unwired.
- 1397
- 1398 The **composite reference** element has the following **child elements**, whatever is not specified is
- 1399 defaulted from the promoted component reference(s).
- 1400 • **interface : Interface (0..1)** - If an **interface** is specified it must provide an interface
- 1401 which is the same or which is a compatible superset of the interface declared by the
- 1402 promoted component reference, i.e. provide a superset of the operations defined by the
- 1403 component for the reference. The interface is described by **zero or one interface**
- 1404 **element** which is a child element of the reference element. For details on the interface
- 1405 element see [the Interface section](#).
- 1406 • **binding : Binding (0..n)** - If one or more **bindings** are specified they **override** any and
- 1407 all of the bindings defined for the promoted component reference from the composite
- 1408 reference perspective. The bindings defined on the component reference are still in effect
- 1409 for local wires within the composite that have the component reference as their source. A
- 1410 reference element has zero or more **binding elements** as children. Details of the binding
- 1411 element are described in the [Bindings section](#). For more details on wiring see [the section](#)
- 1412 [on Wires](#).
- 1413 A reference identifies zero or more target services which satisfy thereference. This can be
- 1414 done in a number of ways, which are fully described in section "5.3.1 Specifying the
- 1415 Target Service(s) for a Reference".
- 1416 • **callback (0..1) / binding : Binding (1..n)** - A **reference** element has an optional
- 1417 **callback** element used if the interface has a callback defined, which has one or more
- 1418 **binding** elements as children. The **callback** and its binding child elements are specified if
- 1419 there is a need to have binding details used to handle callbacks. If the callback element is
- 1420 not present, the behaviour is runtime implementation dependent.
- 1421

1422 6.2.1 Example Reference

1423

1424 The following figure shows the reference symbol that is used to represent a reference in an

1425 assembly diagram.

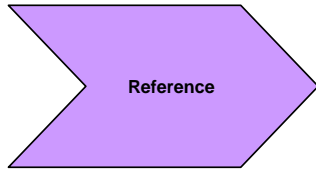


Figure 9: Reference symbol

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

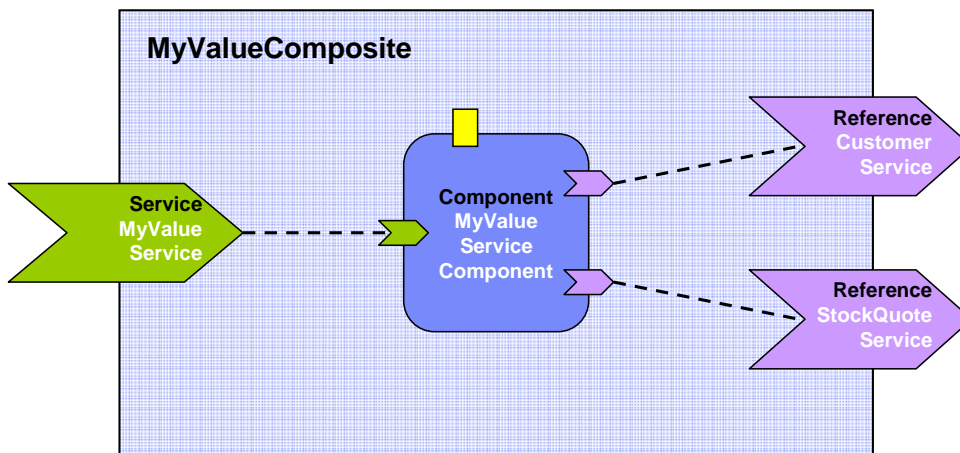


Figure 10: MyValueComposite showing References

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

```
<?xml version="1.0" encoding="ASCII"?>
<!-- MyValueComposite_3 example -->
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
  targetNamespace="http://foo.com"
  name="MyValueComposite" >
  ...
  <component name="MyValueServiceComponent">
```

```

1453         <implementation.java
1454 class="services.myvalue.MyValueServiceImpl"/>
1455         <property name="currency">EURO</property>
1456         <reference name="customerService"/>
1457         <reference name="StockQuoteService"/>
1458     </component>
1459
1460     <reference name="CustomerService"
1461         promote="MyValueServiceComponent/customerService">
1462         <interface.java interface="services.customer.CustomerService"/>
1463         <!-- The following forces the binding to be binding.sca whatever
1464 is -->
1465         <!-- specified by the component reference or by the underlying
1466 -->
1467         <!-- implementation
1468 -->
1469         <binding.sca/>
1470     </reference>
1471
1472     <reference name="StockQuoteService"
1473         promote="MyValueServiceComponent/StockQuoteService">
1474         <interface.java
1475 interface="services.stockquote.StockQuoteService"/>
1476         <binding.ws port="http://www.stockquote.org/StockQuoteService#
1477 wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
1478     </reference>
1479
1480     ...
1481
1482
1483 </composite>
1484

```

1485 6.3 Property

1486 **Properties** allow for the configuration of an implementation with externally set data values. A
1487 composite can declare zero or more properties. Each property has a type, which may be either
1488 simple or complex. An implementation may also define a default value for a property. Properties
1489 are configured with values in the components that use the implementation.

1490 The declaration of a property in a composite follows the form described in the following schema
1491 snippet:

```

1492
1493 <?xml version="1.0" encoding="ASCII"?>
1494 <!-- Composite Property schema snippet -->
1495 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
1496     ...
1497     <property name="xs:NCName" (type="xs:QName" | element="xs:QName")

```

```

        many="xs:boolean"? mustSupply="xs:boolean"?>*
    default-property-value?
</property>
...
</composite>

```

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

- **name : NCName (1..1)** - the name of the property
- one of **(1..1)**:
 - **type : QName** - the type of the property - the qualified name of an XML schema type
 - **element : QName** - the type of the property defined as the qualified name of an XML schema global element - the type is the type of the global element
- **many : boolean (0..1)** - whether the property is single-valued (false) or multi-valued (true). The default is **false**. In the case of a multi-valued property, it is presented to the implementation as a collection of property values.
- **mustSupply : boolean (0..1)** - whether the property value must be supplied by the component that uses the implementation - when mustSupply="true" the component must supply a value since the implementation has no default value for the property. A default-property-value should only be supplied when mustSupply="false" (the default setting for the mustSupply attribute), since the implication of a default value is that it is used only when a value is not supplied by the using component.

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

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

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

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

6.3.1 Property Examples

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

```

<xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://foo.com/"
    xmlns:tns="http://foo.com/">
    <!-- ComplexProperty schema -->
    <xsd:element name="fooElement" type="MyComplexType"/>
    <xsd:complexType name="MyComplexType">

```

```

1543         <xsd:sequence>
1544             <xsd:element name="a" type="xsd:string"/>
1545             <xsd:element name="b" type="anyURI"/>
1546         </xsd:sequence>
1547         <attribute name="attr" type="xsd:string" use="optional"/>
1548     </xsd:complexType>
1549 </xsd:schema>
1550
1551
1552 The following composite demonstrates the declaration of a property of a complex type, with a
1553 default value, plus it demonstrates the setting of a property value of a complex type within a
1554 component:
1555
1556 <?xml version="1.0" encoding="ASCII"?>
1557
1558 <composite      xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1559                xmlns:foo="http://foo.com"
1560                targetNamespace="http://foo.com"
1561                name="AccountServices">
1562 <!-- AccountServices Example1 -->
1563
1564     ...
1565
1566     <property name="complexFoo" type="foo:MyComplexType">
1567         <MyComplexPropertyValue xsi:type="foo:MyComplexType">
1568             <foo:a>AValue</foo:a>
1569             <foo:b>InterestingURI</foo:b>
1570         </MyComplexPropertyValue>
1571     </property>
1572
1573     <component name="AccountServiceComponent">
1574         <implementation.java class="foo.AccountServiceImpl"/>
1575         <property name="complexBar" source="$complexFoo"/>
1576         <reference name="accountDataService"
1577             target="AccountDataServiceComponent"/>
1578         <reference name="stockQuoteService" target="StockQuoteService"/>
1579     </component>
1580
1581     ...
1582 </composite>

```

1582 In the declaration of the property named **complexFoo** in the composite **AccountServices**, the
1583 property is defined to be of type **foo:MyComplexType**. The namespace **foo** is declared in the
1584 composite and it references the example XSD, where MyComplexType is defined. The declaration
1585 of complexFoo contains a default value. This is declared as the content of the property element.
1586 In this example, the default value consists of the element **MyComplexPropertyValue** of type

1587 foo:MyComplexType and its two child elements <foo:a> and <foo:b>, following the definition of
1588 MyComplexType.

1589 In the component **AccountServiceComponent**, the component sets the value of the property
1590 **complexBar**, declared by the implementation configured by the component. In this case, the
1591 type of complexBar is foo:MyComplexType. The example shows that the value of the complexBar
1592 property is set from the value of the complexFoo property – the **source** attribute of the property
1593 element for complexBar declares that the value of the property is set from the value of a property
1594 of the containing composite. The value of the source attribute is **\$complexFoo**, where
1595 complexFoo is the name of a property of the composite. This value implies that the whole of the
1596 value of the source property is used to set the value of the component property.

1597 The following example illustrates the setting of the value of a property of a simple type (a string)
1598 from **part** of the value of a property of the containing composite which has a complex type:

```
1599 <?xml version="1.0" encoding="ASCII"?>
1600
1601 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1602           xmlns:foo="http://foo.com"
1603           targetNamespace="http://foo.com"
1604           name="AccountServices">
1605   <!-- AccountServices Example2 -->
1606
1607   ...
1608
1609   <property name="complexFoo" type="foo:MyComplexType">
1610     <MyComplexPropertyValue xsi:type="foo:MyComplexType">
1611       <foo:a>AValue</foo:a>
1612       <foo:b>InterestingURI</foo:b>
1613     </MyComplexPropertyValue>
1614   </property>
1615
1616   <component name="AccountServiceComponent">
1617     <implementation.java class="foo.AccountServiceImpl"/>
1618     <property name="currency" source="$complexFoo/a"/>
1619     <reference name="accountDataService"
1620       target="AccountDataServiceComponent"/>
1621     <reference name="stockQuoteService" target="StockQuoteService"/>
1622   </component>
1623
1624   ...
1625
1626 </composite>
```

1627 In this example, the component **AccountServiceComponent** sets the value of a property called
1628 **currency**, which is of type string. The value is set from a property of the composite
1629 **AccountServices** using the source attribute set to **\$complexFoo/a**. This is an XPath expression
1630 that selects the property name **complexFoo** and then selects the value of the **a** subelement of
1631 complexFoo. The "a" subelement is a string, matching the type of the currency property.

1632 Further examples of declaring properties and setting property values in a component follow:

1633 Declaration of a property with a simple type and a default value:

```
1634 <property name="SimpleTypeProperty" type="xsd:string">  
1635 MyValue  
1636 </property>
```

1637

1638 Declaration of a property with a complex type and a default value:

```
1639 <property name="complexFoo" type="foo:MyComplexType">  
1640   <MyComplexPropertyValue xsi:type="foo:MyComplexType">  
1641     <foo:a>AValue</foo:a>  
1642     <foo:b>InterestingURI</foo:b>  
1643   </MyComplexPropertyValue>  
1644 </property>
```

1645

1646 Declaration of a property with an element type:

```
1647 <property name="elementFoo" element="foo:fooElement">  
1648   <foo:fooElement>  
1649     <foo:a>AValue</foo:a>  
1650     <foo:b>InterestingURI</foo:b>  
1651   </foo:fooElement>  
1652 </property>
```

1653

1654 Property value for a simple type:

```
1655 <property name="SimpleTypeProperty">  
1656 MyValue  
1657 </property>
```

1658

1659

1660 Property value for a complex type, also showing the setting of an attribute value of the complex
1661 type:

```
1662 <property name="complexFoo">  
1663   <MyComplexPropertyValue xsi:type="foo:MyComplexType" attr="bar">  
1664     <foo:a>AValue</foo:a>  
1665     <foo:b>InterestingURI</foo:b>  
1666   </MyComplexPropertyValue>  
1667 </property>
```

1668

1669 Property value for an element type:

```
1670 <property name="elementFoo">  
1671   <foo:fooElement attr="bar">  
1672     <foo:a>AValue</foo:a>  
1673     <foo:b>InterestingURI</foo:b>  
1674   </foo:fooElement>  
1675 </property>
```

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

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

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

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

6.4 Wire

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

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

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

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

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

```
<?xml version="1.0" encoding="ASCII"?>
<!-- Wires schema snippet -->
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
  targetNamespace="xs:anyURI"
  name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?
  constrainingType="QName"?
  requires="list of xs:QName"? policySets="list of
  xs:QName"?>
```

```

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

```

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

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

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

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

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

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

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

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

Deleted: 23 September

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

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

6.4.1 Wire Examples

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

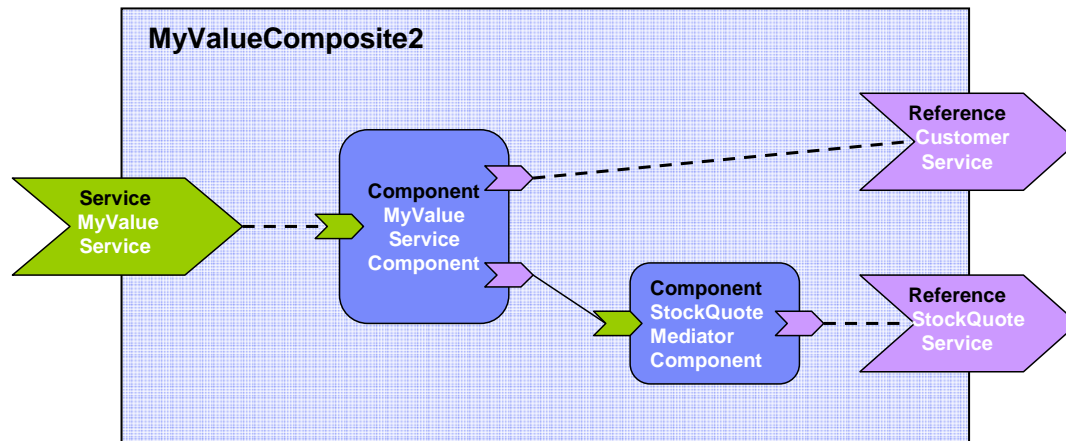


Figure 11: MyValueComposite2 showing Wires

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

```

<?xml version="1.0" encoding="ASCII"?>
<!-- MyValueComposite Wires examples -->
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
  targetNamespace="http://foo.com"
  name="MyValueComposite2" >

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

```

1802         <binding.ws port="http://www.myvalue.org/MyValueService#
1803             wsdl.endpoint(MyValueService/MyValueServiceSOAP)"/>
1804     </service>
1805
1806     <component name="MyValueServiceComponent">
1807         <implementation.java
1808             class="services.myvalue.MyValueServiceImpl"/>
1809         <property name="currency">EURO</property>
1810         <service name="MyValueService"/>
1811         <reference name="customerService"/>
1812         <reference name="stockQuoteService"/>
1813     </component>
1814
1815     <wire source="MyValueServiceComponent/stockQuoteService"
1816         target="StockQuoteMediatorComponent"/>
1817
1818     <component name="StockQuoteMediatorComponent">
1819         <implementation.java class="services.myvalue.SQMediatorImpl"/>
1820         <property name="currency">EURO</property>
1821         <reference name="stockQuoteService"/>
1822     </component>
1823
1824     <reference name="CustomerService"
1825         promote="MyValueServiceComponent/customerService">
1826         <interface.java interface="services.customer.CustomerService"/>
1827         <binding.sca/>
1828     </reference>
1829
1830     <reference name="StockQuoteService"
1831         promote="StockQuoteMediatorComponent">
1832         <interface.java
1833             interface="services.stockquote.StockQuoteService"/>
1834         <binding.ws port="http://www.stockquote.org/StockQuoteService#
1835             wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
1836     </reference>
1837
1838 </composite>
1839

```

6.4.2 Autowire

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

1846 service within the same composite. Autowire works by searching within the composite for a
1847 service interface which matches the interface of the references.

1848 The autowire feature is not used by default. Autowire is enabled by the setting of an autowire
1849 attribute to "true". Autowire is disabled by setting of the autowire attribute to "false" The autowire
1850 attribute can be applied to any of the following elements within a composite:

- 1851 • reference
- 1852 • component
- 1853 • composite

1854 Where an element does not have an explicit setting for the autowire attribute, it inherits the
1855 setting from its parent element. Thus a reference element inherits the setting from its containing
1856 component. A component element inherits the setting from its containing composite. Where
1857 there is no setting on any level, autowire="false" is the default.

1858 As an example, if a composite element has autowire="true" set, this means that autowiring is
1859 enabled for all component references within that composite. In this example, autowiring can be
1860 turned off for specific components and specific references through setting autowire="false" on the
1861 components and references concerned.

1862 For each component reference for which autowire is enabled, the autowire process searches within
1863 the composite for target services which are compatible with the reference. "Compatible" here
1864 means:

- 1865 • the target service interface must be a compatible superset of the reference interface (as
1866 defined in [the section on Wires](#))
- 1867 • the intents, [and policies applied to the service must be compatible on the reference](#) – so
1868 that wiring the reference to the service will not cause an error due to policy mismatch (see
1869 [the Policy Framework specification \[10\]](#) for details)

Comment [mbgl7]: Issue 57

1870 If the search finds **more than 1** valid target service for a particular reference, the action taken
1871 depends on the multiplicity of the reference:

- 1872 • for multiplicity 0..1 and 1..1, the SCA runtime selects one of the target services in a
1873 runtime-dependent fashion and wires the reference to that target service
- 1874 • for multiplicity 0..n and 1..n, the reference is wired to all of the target services

1875 If the search finds **no** valid target services for a particular reference, the action taken depends on
1876 the multiplicity of the reference:

- 1877 • for multiplicity 0..1 and 0..n, there is no problem – no services are wired and there is no
1878 error
- 1879 • for multiplicity 1..1 and 1..n, an error is raised by the SCA runtime since the reference is
1880 intended to be wired

1881

1882 6.4.3 Autowire Examples

1883 This example demonstrates two versions of the same composite – the first version is done using
1884 explicit wires, with no autowiring used, the second version is done using autowire. In both cases
1885 the end result is the same – the same wires connect the references to the services.

1886 First, here is a diagram for the composite:

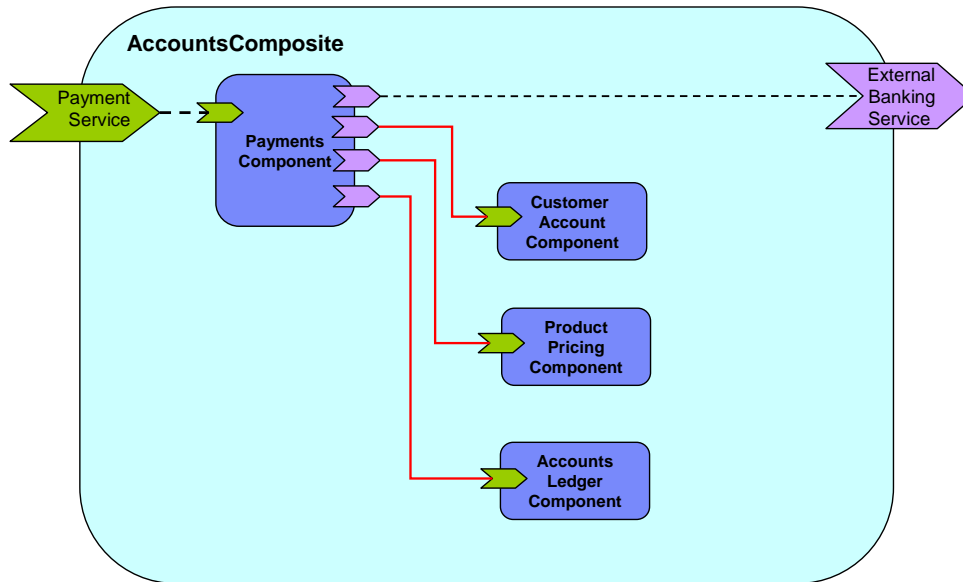


Figure 12: Example Composite for Autowire

First, the composite using explicit wires:

```

<?xml version="1.0" encoding="UTF-8"?>
<!-- Autowire Example - No autowire -->
<composite xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
  xmlns:foo="http://foo.com"
  targetNamespace="http://foo.com"
  name="AccountComposite">

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

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

  <component name="CustomerAccountComponent">

```

```

1913         <implementation.java class="com.foo.accounts.CustomerAccount"/>
1914     </component>
1915
1916     <component name="ProductPricingComponent">
1917         <implementation.java class="com.foo.accounts.ProductPricing"/>
1918     </component>
1919
1920     <component name="AccountsLedgerComponent">
1921         <implementation.composite name="foo:AccountsLedgerComposite"/>
1922     </component>
1923
1924     <reference name="ExternalBankingService"
1925         promote="PaymentsComponent/ExternalBankingService"/>
1926
1927 </composite>
1928

```

Secondly, the composite using autowire:

```

1930 <?xml version="1.0" encoding="UTF-8"?>
1931 <!-- Autowire Example - With autowire -->
1932 <composite xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
1933     xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1934     xmlns:foo="http://foo.com"
1935     targetNamespace="http://foo.com"
1936     name="AccountComposite">
1937
1938     <service name="PaymentService" promote="PaymentsComponent">
1939         <interface.java class="com.foo.PaymentServiceInterface"/>
1940     </service>
1941
1942     <component name="PaymentsComponent" autowire="true">
1943         <implementation.java class="com.foo.accounts.Payments"/>
1944         <service name="PaymentService"/>
1945         <reference name="CustomerAccountService"/>
1946         <reference name="ProductPricingService"/>
1947         <reference name="AccountsLedgerService"/>
1948         <reference name="ExternalBankingService"/>
1949     </component>
1950
1951     <component name="CustomerAccountComponent">
1952         <implementation.java class="com.foo.accounts.CustomerAccount"/>
1953     </component>
1954
1955     <component name="ProductPricingComponent">

```



```

1956         <implementation.java class="com.foo.accounts.ProductPricing"/>
1957     </component>
1958
1959     <component name="AccountsLedgerComponent">
1960         <implementation.composite name="foo:AccountsLedgerComposite"/>
1961     </component>
1962
1963     <reference name="ExternalBankingService"
1964         promote="PaymentsComponent/ExternalBankingService"/>
1965
1966 </composite>

```

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

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

6.5 Using Composites as Component Implementations

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

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

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

- the composite must have at least one service or at least one reference.
A component with no services and no references is not meaningful in terms of SCA, since it cannot be wired to anything – it neither provides nor consumes any services
- each service offered by the composite must be wired to a service of a component or to a composite reference.
If services are left unwired, the implication is that some exception will occur at runtime if the service is invoked.

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

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

```

1998
1999 <?xml version="1.0" encoding="ASCII"?>
2000 <!-- Composite Implementation schema snippet -->
2001 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2002     targetNamespace="xs:anyURI"

```

```

2003         name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?
2004         constrainingType="QName"?
2005         requires="list of xs:QName"? policySets="list of
2006 xs:QName"?>
2007
2008     ...
2009
2010     <component name="xs:NCName" autowire="xs:boolean"?
2011         requires="list of xs:QName"? policySets="list of xs:QName"?>*
2012         <implementation.composite name="xs:QName"/>?
2013         <service name="xs:NCName" requires="list of xs:QName"?
2014             policySets="list of xs:QName"?>*
2015             <interface ... />?
2016             <binding uri="xs:anyURI" name="xs:QName"?
2017                 requires="list of xs:QName"
2018                 policySets="list of xs:QName"?/>*
2019             <callback>?
2020                 <binding uri="xs:anyURI"? name="xs:QName"?
2021                     requires="list of xs:QName"?
2022                     policySets="list of xs:QName"?/>+
2023             </callback>
2024         </service>
2025         <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
2026             source="xs:string"? file="xs:anyURI"?>*
2027             property-value
2028         </property>
2029         <reference name="xs:NCName" target="list of xs:anyURI"?
2030             autowire="xs:boolean"? wiredByImpl="xs:boolean"?
2031             requires="list of xs:QName"? policySets="list of xs:QName"?
2032             multiplicity="0..1 or 1..1 or 0..n or 1..n"?/>*
2033         <interface ... />?
2034         <binding uri="xs:anyURI"? name="xs:QName"?
2035             requires="list of xs:QName" policySets="list of
2036 xs:QName"?/>*
2037         <callback>?
2038             <binding uri="xs:anyURI"? name="xs:QName"?
2039                 requires="list of xs:QName"?
2040                 policySets="list of xs:QName"?/>+
2041         </callback>
2042     </reference>
2043 </component>
2044
2045     ...

```

</composite>

The implementation.composite element has the following attribute:

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

6.5.1 Example of Composite used as a Component Implementation

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

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- CompositeComponent example -->
<composite xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
  xsd:schemaLocation="http://docs.oasis-open.org/ns/opencsa/sca/200712
file:/C:/Strategy/SCA/v09_osoaschemas/schemas/sca.xsd"
  xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
  targetNamespace="http://foo.com"
  xmlns:foo="http://foo.com"
  name="AccountComposite">

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

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

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

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

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

```

2089     <reference name="StockQuoteService"/>
2090
2091     <property name="currency" source="$currency"/>
2092 </component>
2093
2094 <component name="AccountDataService">
2095     <implementation.composite name="foo:AccountDataServiceComposite"/>
2096
2097     <property name="currency" source="$currency"/>
2098 </component>
2099
2100 </composite>
2101

```

6.6 Using Composites through Inclusion

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

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

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

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

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

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

```

2125 <?xml version="1.0" encoding="UTF-8"?>
2126 <!-- Include snippet -->
2127 <composite      xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2128                targetNamespace="xs:anyURI"
2129                name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?
2130                constrainingType="QName"?
2131                requires="list of xs:QName"? policySets="list of
2132 xs:QName"?>
2133
2134     ...

```

```

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

```

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

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

6.6.1 Included Composite Examples

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

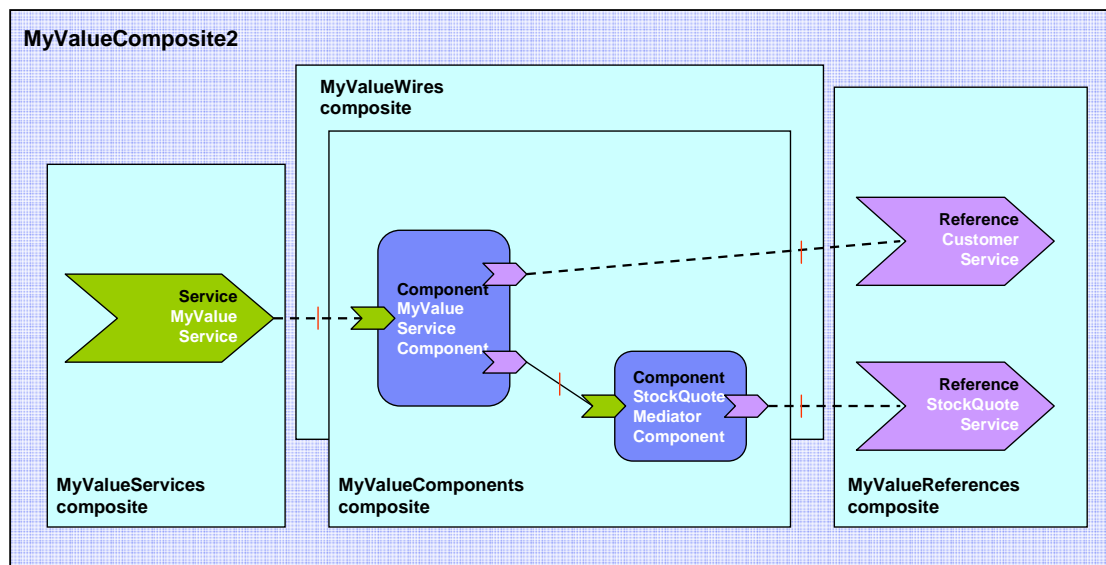


Figure 13 MyValueComposite2 built from 4 included composites

The following snippet shows the contents of the MyValueComposite2.composite file for the MyValueComposite2 built using included composites. In this sample it only provides the name of

the composite. The composite file itself could be used in a scenario using included composites to define components, services, references and wires.

```
<?xml version="1.0" encoding="ASCII"?>
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
  targetNamespace="http://foo.com"
  xmlns:foo="http://foo.com"
  name="MyValueComposite2" >

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

</composite>
```

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

```
<?xml version="1.0" encoding="ASCII"?>
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
  targetNamespace="http://foo.com"
  xmlns:foo="http://foo.com"
  name="MyValueServices" >

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

</composite>
```

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

```
<?xml version="1.0" encoding="ASCII"?>
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
  targetNamespace="http://foo.com"
  xmlns:foo="http://foo.com"
  name="MyValueComponents" >

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

</composite>
```

```

2207         <property name="currency">EURO</property>
2208     </component>
2209
2210     <component name="StockQuoteMediatorComponent">
2211         <implementation.java class="services.myvalue.SQMediatorImpl"/>
2212         <property name="currency">EURO</property>
2213     </component>
2214
2215 </composite>
2216

```

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

```

2217
2218
2219 <?xml version="1.0" encoding="ASCII"?>
2220 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2221     targetNamespace="http://foo.com"
2222     xmlns:foo="http://foo.com"
2223     name="MyValueReferences" >
2224
2225     <reference name="CustomerService"
2226         promote="MyValueServiceComponent/CustomerService">
2227         <interface.java interface="services.customer.CustomerService"/>
2228         <binding.sca/>
2229     </reference>
2230
2231     <reference name="StockQuoteService"
2232     promote="StockQuoteMediatorComponent">
2233         <interface.java
2234     interface="services.stockquote.StockQuoteService"/>
2235         <binding.ws port="http://www.stockquote.org/StockQuoteService#
2236     wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
2237     </reference>
2238
2239 </composite>

```

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

```

2240
2241
2242 <?xml version="1.0" encoding="ASCII"?>
2243 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2244     targetNamespace="http://foo.com"
2245     xmlns:foo="http://foo.com"
2246     name="MyValueWires" >
2247
2248     <wire source="MyValueServiceComponent/stockQuoteService"
2249         target="StockQuoteMediatorComponent"/>

```

2250
2251 </composite>

2252 **6.7 Composites which Include Component Implementations of**
2253 **Multiple Types**

2254
2255 A Composite containing multiple components MAY have multiple component implementation types.
2256 For example, a Composite may include one component with a Java POJO as its implementation
2257 and another component with a BPEL process as its implementation.
2258

7 ConstrainingType

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

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

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

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

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

```
<?xml version="1.0" encoding="ASCII"?>
<!-- ConstrainingType schema snippet -->
<constrainingType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
    targetNamespace="xs:anyURI"
    name="xs:NCName" requires="list of xs:QName"?>

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

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

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

</constrainingType>

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

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

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

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

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

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

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

7.1 Example constrainingType

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

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

```

2352     <reference name="StockQuoteService"
2353         target="StockQuoteMediatorComponent" />
2354 </component>
2355
2356 <constrainingType name="CT"
2357     targetNamespace="http://myns.com">
2358     <service name="MyValueService">
2359         <interface.java interface="services.myvalue.MyValueService" />
2360     </service>
2361     <reference name="customerService">
2362         <interface.java interface="services.customer.CustomerService" />
2363     </reference>
2364     <reference name="stockQuoteService">
2365         <interface.java interface="services.stockquote.StockQuoteService" />
2366     </reference>
2367     <property name="currency" type="xsd:string" />
2368 </constrainingType>

```

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

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

8 Interface

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

SCA currently supports the following interface type systems:

- Java interfaces
- WSDL 1.1 portTypes ([Web Services Definition Language \[8\]](#))
- WSDL 2.0 interfaces ([Web Services Definition Language \[8\]](#))
- C++ classes

Comment [mbgl8]: Issue 69 part 2

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

The following snippet shows the definition for the **interface** base element.

Comment [mbgl9]: Issue 39

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

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

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

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

```
<interface.wSDL interface="xs:anyURI" ... />
```

The interface.wSDL element has the following attributes:

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

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

```
<interface.wSDL interface="http://www.stockquote.org/StockQuoteService#  
wSDL.interface(StockQuote)" />
```

For WSDL 1.1, the interface attribute points to a portType in the WSDL. For WSDL 2.0, the interface attribute points to an interface in the WSDL. For the WSDL 1.1 portType and WSDL 2.0

Deleted: 23 September

2417 interface type systems, arguments and return of the service operations are described using XML
2418 schema.

2419 For information about Java interfaces, including details of SCA-specific annotations, see the SCA
2420 Java Common Annotations and APIs specification [1].

2421 8.1 Local and Remotable Interfaces

2422 A remotable service is one which may be called by a client which is running in an operating system
2423 process different from that of the service itself (this also applies to clients running on different
2424 machines from the service). Whether a service of a component implementation is remotable is
2425 defined by the interface of the service. In the case of Java this is defined by adding the
2426 **@Remotable** annotation to the Java interface (see [Client and Implementation Model Specification](#)
2427 [for Java](#)). WSDL defined interfaces are always remotable.

2428
2429 The style of remotable interfaces is typically **coarse grained** and intended for **loosely coupled**
2430 interactions. Remotable service Interfaces MUST NOT make use of **method or operation**
2431 **overloading**.

2432
2433 Independent of whether the remotable service is called remotely from outside the process where
2434 the service runs or from another component running in the same process, the data exchange
2435 semantics are **by-value**.

2436 Implementations of remotable services may modify input messages (parameters) during or after
2437 an invocation and may modify return messages (results) after the invocation. If a remotable
2438 service is called locally or remotely, the SCA container is responsible for making sure that no
2439 modification of input messages or post-invocation modifications to return messages are seen by
2440 the caller.

2441 Here is a snippet which shows an example of a remotable java interface:

```
2442  
2443 package services.hello;  
2444  
2445 @Remotable  
2446 public interface HelloService {  
2447  
2448     String hello(String message);  
2449 }  
2450
```

2451 It is possible for the implementation of a remotable service to indicate that it can be called using
2452 by-reference data exchange semantics when it is called from a component in the same process.
2453 This can be used to improve performance for service invocations between components that run in
2454 the same process. This can be done using the @AllowsPassByReference annotation (see the [Java](#)
2455 [Client and Implementation Specification](#)).

2456
2457 A service typed by a local interface can only be called by clients that are running in the same
2458 process as the component that implements the local service. Local services cannot be published
2459 via remotable services of a containing composite. In the case of Java a local service is defined by a
2460 Java interface definition without a **@Remotable** annotation.

2461
2462 The style of local interfaces is typically **fine grained** and intended for **tightly coupled**
2463 interactions. Local service interfaces can make use of **method or operation overloading**.

2464 The data exchange semantic for calls to services typed by local interfaces is **by-reference**.

Deleted: 23 September

2465

2466 8.2 Bidirectional Interfaces

2467 The relationship of a business service to another business service is often peer-to-peer, requiring
2468 a two-way dependency at the service level. In other words, a business service represents both a
2469 consumer of a service provided by a partner business service and a provider of a service to the
2470 partner business service. This is especially the case when the interactions are based on
2471 asynchronous messaging rather than on remote procedure calls. The notion of **bidirectional**
2472 **interfaces** is used in SCA to directly model peer-to-peer bidirectional business service
2473 relationships.

2474 An interface element for a particular interface type system must allow the specification of an
2475 optional callback interface. If a callback interface is specified SCA refers to the interface as a whole
2476 as a bidirectional interface.

2477 The following snippet shows the interface element defined using Java interfaces with an optional
2478 callbackInterface attribute.

2479

```
2480 <interface.java          interface="services.invoicing.ComputePrice"  
2481                        callbackInterface="services.invoicing.InvoiceCallback"/>
```

2482

2483 If a service is defined using a bidirectional interface element then its implementation implements
2484 the interface, and its implementation uses the callback interface to converse with the client that
2485 called the service interface.

2486

2487 If a reference is defined using a bidirectional interface element, the client component
2488 implementation using the reference calls the referenced service using the interface. The client
2489 must provide an implementation of the callback interface.

2490 Callbacks may be used for both remotable and local services. Either both interfaces of a
2491 bidirectional service MUST be remotable, or both MUST be local. A bidirectional service MUST NOT
2492 mix local and remote services.

2493

2494 8.3 Conversational Interfaces

2495

2496 Services sometimes cannot easily be defined so that each operation stands alone and is
2497 completely independent of the other operations of the same service. Instead, there is a sequence
2498 of operations that must be called in order to achieve some higher level goal. SCA calls this
2499 sequence of operations a **conversation**. If the service uses a bidirectional interface, the
2500 conversation may include both operations and callbacks.

2501

2502 Such conversational services are typically managed by using conversation identifiers that are
2503 either (1) part of the application data (message parts or operation parameters) or 2)
2504 communicated separately from application data (possibly in headers). SCA introduces the concept
2505 of *conversational interfaces* for describing the interface contract for conversational services of the
2506 second form above. With this form, it is possible for the runtime to automatically manage the
2507 conversation, with the help of an appropriate binding specified at deployment. SCA does not
2508 standardize any aspect of conversational services that are maintained using application data.
2509 Such services are neither helped nor hindered by SCA's conversational service support.

2510

2511 Conversational services typically involve state data that relates to the conversation that is taking
2512 place. The creation and management of the state data for a conversation has a significant impact
2513 on the development of both clients and implementations of conversational services.

2514

2515 Traditionally, application developers who have needed to write conversational services have been
2516 required to write a lot of plumbing code. They need to:

2517

- 2518 - choose or define a protocol to communicate conversational (correlation) information
2519 between the client & provider
- 2520 - route conversational messages in the provider to a machine that can handle that
2521 conversation, while handling concurrent data access issues
- 2522 - write code in the client to use/encode the conversational information
- 2523 - maintain state that is specific to the conversation, sometimes persistently and
2524 transactionally, both in the implementation and the client.

2525

2526 SCA makes it possible to divide the effort associated with conversational services between a
2527 number of roles:

- 2528 - Application Developer: Declares that a service interface is conversational (leaving the
2529 details of the protocol up to the binding). Uses lifecycle semantics, APIs or other
2530 programmatic mechanisms (as defined by the implementation-type being used) to
2531 manage conversational state.
- 2532 - Application Assembler: chooses a binding that can support conversations
- 2533 - Binding Provider: implements a protocol that can pass conversational information with
2534 each operation request/response.
- 2535 - Implementation-Type Provider: defines APIs and/or other programmatic mechanisms for
2536 application developers to access conversational information. Optionally implements
2537 instance lifecycle semantics that automatically manage implementation state based on
2538 the binding's conversational information.

2539

2540 There is a policy intent with the name conversational which is used to mark an interface as being
2541 conversational in nature. Where a service or a reference has a conversational interface, the
2542 conversational intent MUST be attached either to the interface itself, or to the service or reference
2543 using the interface. How to attach the conversational intent to an interface depends on the type of
2544 the interface. For a WSDL interface, this is described in section 8.4 "SCA-Specific Aspects for
2545 WSDL Interfaces". For a Java interface, it is described in the Java Common Annotations and APIs
2546 specification. Note that setting the conversational intent on the service or reference element is
2547 useful when reusing an existing interface definition that contains no SCA information, since it
2548 requires no modification of the interface artifact.

2549

2550 The meaning of the conversational intent is that both the client and the provider of the interface
2551 may assume that messages (in either direction) will be handled as part of an ongoing conversation
2552 without depending on identifying information in the body of the message (i.e. in parameters of the
2553 operations). In effect, the conversation interface specifies a high-level abstract protocol that must
2554 be satisfied by any actual binding/policy combination used by the service.

2555 Examples of binding/policy combinations that support conversational interfaces are:

- 2556 - Web service binding with a WS-RM policy
- 2557 - Web service binding with a WS-Addressing policy
- 2558 - Web service binding with a WS-Context policy

Comment [mbgl10]: Issue
35

Comment [mbgl11]: Need
proper section link here

Deleted: 23 September

2559 - JMS binding with a conversation policy that uses the JMS correlationID header
 2560

2561 Conversations occur between one client and one target service. Consequently, requests originating
 2562 from one client to multiple target conversational services will result in multiple conversations. For
 2563 example, if a client A calls services B and C, both of which implement conversational interfaces,
 2564 two conversations result, one between A and B and another between A and C. Likewise, requests
 2565 flowing through multiple implementation instances will result in multiple conversations. For
 2566 example, a request flowing from A to B and then from B to C will involve two conversations (A and
 2567 B, B and C). In the previous example, if a request was then made from C to A, a third
 2568 conversation would result (and the implementation instance for A would be different from the one
 2569 making the original request).

2570 Invocation of any operation of a conversational interface MAY start a conversation. The decision on
 2571 whether an operation would start a conversation depends on the component's implementation and
 2572 its implementation type. Implementation types MAY support components with conversational
 2573 services. If an implementation type does provide this support, it must provide a mechanism for
 2574 determining when a new conversation should be used for an operation (for example, in Java, the
 2575 conversation is new on the first use of an injected reference; in BPEL, the conversation is new
 2576 when the client's partnerLink comes into scope).

2577

2578 One or more operations in a conversational interface may be annotated with an *endsConversation*
 2579 annotation (the mechanism for annotating the interface depends on the interface type). Where an
 2580 interface is **bidirectional**, operations may also be annotated in this way on operations of a
 2581 callback interface. When a conversation ending operation is called, it indicates to both the client
 2582 and the service provider that the conversation is complete. Any subsequent attempts to call an
 2583 operation or a callback operation associated with the same conversation will generate a
 2584 `sca:ConversationViolation` fault.

2585 A `sca:ConversationViolation` fault is thrown when one of the following errors occur:

- 2586 - A message is received for a particular conversation, after the conversation has ended
- 2587 - The conversation identification is invalid (not unique, out of range, etc.)
- 2588 - The conversation identification is not present in the input message of the operation that
 2589 ends the conversation
- 2590 - The client or the service attempts to send a message in a conversation, after the
 2591 conversation has ended

2592 This fault is named within the SCA namespace standard prefix "sca", which corresponds to URI
 2593 <http://docs.oasis-open.org/ns/opencsa/sca/200712>.

2594 The lifecycle of resources and the association between unique identifiers and conversations are
 2595 determined by the service's implementation type and may not be directly affected by the
 2596 "endConversation" annotation. For example, a **WS-BPEL** process may outlive most of the
 2597 conversations that it is involved in.

2598 Although conversational interfaces do not require that any identifying information be passed as
 2599 part of the body of messages, there is conceptually an identity associated with the conversation.
 2600 Individual implementations types MAY provide an API to access the ID associated with the
 2601 conversation, although no assumptions may be made about the structure of that identifier.
 2602 Implementation types MAY also provide a means to set the conversation ID by either the client or
 2603 the service provider, although the operation may only be supported by some binding/policy
 2604 combinations.

2605

2606 Implementation-type specifications are encouraged to define and provide conversational instance
 2607 lifecycle management for components that implement conversational interfaces. However,
 2608 implementations may also manage the conversational state manually.

2609

8.4 SCA-Specific Aspects for WSDL Interfaces

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

For WSDL interfaces, there is an extension mechanism that permits additional information to be included within the WSDL document. SCA takes advantage of this extension mechanism. In order to use the SCA extension mechanism, the SCA namespace (<http://docs.oasis-open.org/ns/opencsa/sca/200712>) must be declared within the WSDL document.

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

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

```
<simpleType name="listOfQNames">
```

```
<list itemType="QName"/>
```

```
</simpleType>
```

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

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

```
requires="conversational"
```

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

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

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

```
...
```

```
<portType name="LoanService" sca:requires="conversational">
```

```
<operation name="apply">
```

```
<input message="tns:ApplicationInput"/>
```

```
<output message="tns:ApplicationOutput"/>
```

```
</operation>
```

```
<operation name="cancel" sca:endsConversation="true">
```

```
</operation>
```

```
...
```

```
</portType>
```

```
...
```

9 Binding

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

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

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

```
<?xml version="1.0" encoding="ASCII"?>
<!-- Bindings schema snippet -->
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
            targetNamespace="xs:anyURI"
            name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?
            constrainingType="QName"?
            requires="list of xs:QName"? policySets="list of
xs:QName"?>
    ...

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

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

```

2696         promote="list of xs:anyURI"? wiredByImpl="xs:boolean"?
2697         multiplicity="0..1 or 1..1 or 0..n or 1..n"?
2698         requires="list of xs:QName"? policySets="list of xs:QName"?>*
2699     <interface ... />?
2700     <binding uri="xs:anyURI"? name="xs:NCName"?
2701         requires="list of xs:QName"? policySets="list of
2702 xs:QName"?/>*
2703     <callback>?
2704         <binding uri="xs:anyURI"? name="xs:NCName"?
2705             requires="list of xs:QName"?
2706             policySets="list of xs:QName"?/>+
2707     </callback>
2708 </reference>
2709
2710 ...
2711
2712 </composite>
2713

```

2714 The element name of the binding element is architected; it is in itself a qualified name. The first
2715 qualifier is always named "binding", and the second qualifier names the respective binding-type
2716 (e.g. binding.composite, binding.ws, binding.ejb, binding.eis).

2717
2718 A binding element has the following attributes:

- 2719 • **uri (optional)** - has the following semantic.
 - 2720 ○ The uri attribute can be omitted.
 - 2721 ○ For a binding of a **reference** the URI attribute defines the target URI of the
2722 reference. This MUST be either the componentName/serviceName for a wire to an
2723 endpoint within the SCA domain, or the accessible address of some service
2724 endpoint either inside or outside the SCA domain (where the addressing scheme is
2725 defined by the type of the binding).
 - 2726 ○ The circumstances under which the uri attribute can be used are defined in
2727 section "5.3.1 Specifying the Target Service(s) for a Reference."
 - 2728 ○ For a binding of a **service** the URI attribute defines the URI relative to the
2729 component, which contributes the service to the SCA domain. The default value for
2730 the URI is the value of the name attribute of the binding.
- 2731 • **name (optional)** – a name for the binding instance (an NCName). The name attribute
2732 allows distinction between multiple binding elements on a single service or reference. The
2733 default value of the name attribute is the service or reference name. When a service or
2734 reference has multiple bindings, only one can have the default value; all others must have
2735 a value specified that is unique within the service or reference. The name also permits the
2736 binding instance to be referenced from elsewhere – particularly useful for some types of
2737 binding, which can be declared in a definitions document as a template and referenced
2738 from other binding instances, simplifying the definition of more complex binding instances
2739 (see [the JMS Binding specification \[11\]](#) for examples of this referencing).
- 2740 • **requires (optional)** - a list of policy intents. See the [Policy Framework specification \[10\]](#)
2741 for a description of this attribute.
- 2742 • **policySets (optional)** – a list of policy sets. See the [Policy Framework specification \[10\]](#)
2743 for a description of this attribute.

2744 When multiple bindings exist for an service, it means that the service is available by any of the
2745 specified bindings. The technique that the SCA runtime uses to choose among available bindings
2746 is left to the implementation and it may include additional (nonstandard) configuration. Whatever
2747 technique is used SHOULD be documented.

2748 Services and References can always have their bindings overridden at the SCA domain level,
2749 unless restricted by Intents applied to them.

2750

2751 If a reference has any bindings they must be "resolved". The bindings MUST include a value for
2752 the @URI or must otherwise specify an endpoint. The reference MUST NOT be wired using SCA
2753 mechanisms. To specify constraints on the kinds of bindings that are acceptable for use with a
2754 reference, the user should specify either policy intents or policy sets.

Comment [mbgl12]: Issue
57

2755
2756 Users may also specifically wire, not just to a component service, but to a specific binding offered
2757 by that target service. To do so, a wire target may optionally be specified with a syntax of
2758 "componentName/serviceName/bindingName".

2759

2760 The following sections describe the SCA and Web service binding type in detail.

2761

2762 9.1 Messages containing Data not defined in the Service Interface

2763

2764 It is possible for a message to include information that is not defined in the interface used to
2765 define the service, for instance information may be contained in SOAP headers or as MIME
2766 attachments.

2767 Implementation types MAY make this information available to component implementations in their
2768 execution context. These implementation types must indicate how this information is accessed
2769 and in what form they are presented.

2770

2771 9.2 Form of the URI of a Deployed Binding

2772

2773 9.2.1 Constructing Hierarchical URIs

2774 Bindings that use hierarchical URI schemes construct the effective URI with a combination of the
2775 following pieces:

2776 Base System URI for a scheme / Component URI / Service Binding URI

2777

2778 Each of these components deserves addition definition:

2779 **Base Domain URI for a scheme.** An SCA domain should define a base URI for each hierarchical
2780 URI scheme on which it intends to provide services.

2781 For example: the HTTP and HTTPS schemes would each have their own base URI defined for the
2782 domain. An example of a scheme that is not hierarchical, and therefore will have no base URI is
2783 the "jms:" scheme.

2784 **Component URI.** The component URI above is for a component that is deployed in the SCA
2785 Domain. The URI of a component defaults to the name of the component, which is used as a
2786 relative URI. The component may have a specified URI value. The specified URI value may be an
2787 absolute URI in which case it becomes the Base URI for all the services belonging to the
2788 component. If the specified URI value is a relative URI, it is used as the Component URI value
2789 above.

Deleted: 23 September

2790 **Service Binding URI.** The Service Binding URI is the relative URI specified in the "uri" attribute
 2791 of a binding element of the service. The default value of the attribute is value of the binding's
 2792 name attribute treated as a relative URI. If multiple bindings for a single service use the same
 2793 scheme (e.g. HTTP), then only one of the bindings may depend on the default value for the uri
 2794 attribute, i.e. only one may use the default binding name. The service binding URI may also be
 2795 absolute, in which case the absolute URI fully specifies the full URI of the service. Some
 2796 deployment environments may not support the use of absolute URIs in service bindings.

2797 Services deployed into the Domain (as opposed to services of components) have a URI that does
 2798 not include a component name, i.e.:

2799 Base Domain URI for a scheme / Service Binding URI

2800 The name of the containing composite does not contribute to the URI of any service.

2801 For example, a service where the Base URI is "http://acme.com", the component is named
 2802 "stocksComponent" and the service binding name is "getQuote", the URI would look like this:

2803 http://acme.com/stocksComponent/getQuote

2804 Allowing a binding's relative URI to be specified that differs from the name of the service allows
 2805 the URI hierarchy of services to be designed independently of the organization of the domain.

2806 It is good practice to design the URI hierarchy to be independent of the domain organization, but
 2807 there may be times when domains are initially created using the default URI hierarchy. When this
 2808 is the case, the organization of the domain can be changed, while maintaining the form of the URI
 2809 hierarchy, by giving appropriate values to the **uri** attribute of select elements. Here is an example
 2810 of a change that can be made to the organization while maintaining the existing URIs:

2811 To move a subset of the services out of one component (say "foo") to a new component (say
 2812 "bar"), the new component should have bindings for the moved services specify a URI
 2813 "../foo/MovedService"..

2814 The URI attribute may also be used in order to create shorter URIs for some endpoints, where the
 2815 component name may not be present in the URI at all. For example, if a binding has a **uri**
 2816 attribute of "../myService" the component name will not be present in the URI.

2817 9.2.2 Non-hierarchical URIs

2818 Bindings that use non-hierarchical URI schemes (such as jms: or mailto:) may optionally make
 2819 use of the "uri" attribute, which is the complete representation of the URI for that service
 2820 binding. Where the binding does not use the "uri" attribute, the binding must offer a different
 2821 mechanism for specifying the service address.

2822 9.2.3 Determining the URI scheme of a deployed binding

2823 One of the things that needs to be determined when building the effective URI of a deployed
 2824 binding (i.e. endpoint) is the URI scheme. The process of determining the endpoint URI scheme is
 2825 binding type specific.

2826 If the binding type supports a single protocol then there is only one URI scheme associated with it.
 2827 In this case, that URI scheme is used.

2828 If the binding type supports multiple protocols, the binding type implementation determines the
 2829 URI scheme by introspecting the binding configuration, which may include the policy sets
 2830 associated with the binding.

2831 A good example of a binding type that supports multiple protocols is binding.ws, which can be
 2832 configured by referencing either an "abstract" WSDL element (i.e. portType or interface) or a
 2833 "concrete" WSDL element (i.e. binding, port or endpoint). When the binding references a PortType
 2834 or Interface, the protocol and therefore the URI scheme is derived from the intents/policy sets
 2835 attached to the binding. When the binding references a "concrete" WSDL element, there are two
 2836 cases:

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

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

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

9.3 SCA Binding

The SCA binding element is defined by the following schema.

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

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

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

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

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

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

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

9.3.1 Example SCA Binding

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

```
<?xml version="1.0" encoding="ASCII"?>
<!-- Binding SCA example -->
```

```

2883 <composite      xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2884               targetNamespace="http://foo.com"
2885               name="MyValueComposite" >
2886
2887     <service name="MyValueService" promote="MyValueComponent">
2888       <interface.java interface="services.myvalue.MyValueService"/>
2889       <binding.sca/>
2890       ...
2891     </service>
2892
2893     ...
2894
2895     <reference name="StockQuoteService"
2896     promote="MyValueComponent/StockQuoteReference">
2897       <interface.java
2898       interface="services.stockquote.StockQuoteService"/>
2899       <binding.sca/>
2900     </reference>
2901
2902 </composite>
2903

```

2904 9.4 Web Service Binding

2905 SCA defines a Web services binding. This is described in [a separate specification document \[9\]](#).

2906

2907 9.5 JMS Binding

2908 SCA defines a JMS binding. This is described in [a separate specification document \[11\]](#).

10 SCA Definitions

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

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

```
<?xml version="1.0" encoding="ASCII"?>
<!-- Composite schema snippet -->
<definitions xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
             targetNamespace="xs:anyURI">

    <sca:intent/>*

    <sca:policySet/>*

    <sca:binding/>*

    <sca:bindingType/>*

    <sca:implementationType/>*

</definitions>
```

The definitions element has the following attribute:

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

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

11 Extension Model

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

The SCA Client and Implementation specifications and the SCA Bindings specifications (see [1], [9], [11]) use these XML Schema substitution groups to define some basic types of interfaces, implementations and bindings, but other types can be defined as required, where support for these extra ones is available from the runtime. The interface type elements, implementation type elements, and binding type elements defined by the SCA specifications are all part of the SCA namespace ("http://docs.oasis-open.org/ns/opencsa/sca/200712"), as indicated in their respective schemas. New interface types, implementation types and binding types that are defined using this extensibility model, which are not part of these SCA specifications must be defined in namespaces other than the SCA namespace.

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

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

11.1 Defining an Interface Type

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

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2006 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
  xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
  elementFormDefault="qualified">
  ...

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

Comment [mbgl13]: Issue 39

Deleted: 23 September

2985
2986
2987
2988

...

</schema>

2989 In the following snippet we show how the base definition is extended to support Java interfaces.
2990 The snippet shows the definition of the **interface.java** element and the **JavaInterface** type
2991 contained in **sca-interface-java.xsd**.

2992
2993
2994
2995
2996
2997
2998

<?xml version="1.0" encoding="UTF-8"?>

<schema xmlns="http://www.w3.org/2001/XMLSchema"

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

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

<element name="interface.java" type="sca:JavaInterface"

substitutionGroup="sca:interface" />

<complexType name="JavaInterface">

<complexContent>

<extension base="sca:Interface">

use="required" />
<attribute name="interface" type="NCName"

</extension>

</complexContent>

</complexType>

</schema>

3010 In the following snippet we show an example of how the base definition can be extended by other
3011 specifications to support a new interface not defined in the SCA specifications. The snippet shows
3012 the definition of the **my-interface-extension** element and the **my-interface-extension-type**
3013 type.

3014
3015
3016
3017
3018
3019

<?xml version="1.0" encoding="UTF-8"?>

<schema xmlns="http://www.w3.org/2001/XMLSchema"

targetNamespace="http://www.example.org/myextension"

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

xmlns:tns="http://www.example.org/myextension">

<element name="my-interface-extension" type="tns:my-interface-
extension-type"

substitutionGroup="sca:interface" />

<complexType name="my-interface-extension-type">

<complexContent>

<extension base="sca:Interface">

...

</extension>

</complexContent>

</complexType>

3030 </schema>
3031

3032 11.2 Defining an Implementation Type

3033 The following snippet shows the base definition for the **implementation** element and
3034 **Implementation** type contained in **sca-core.xsd**; see appendix for complete schema.

```
3035  
3036       <?xml version="1.0" encoding="UTF-8"?>  
3037       <!-- (c) Copyright SCA Collaboration 2006 -->  
3038       <schema xmlns="http://www.w3.org/2001/XMLSchema"  
3039               targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
3040               xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
3041               elementFormDefault="qualified">  
3042  
3043               ...  
3044  
3045               <element name="implementation" type="sca:Implementation"  
3046               abstract="true"/>  
3047               <complexType name="Implementation"/>  
3048  
3049               ...  
3050  
3051       </schema>
```

3052
3053 In the following snippet we show how the base definition is extended to support Java
3054 implementation. The snippet shows the definition of the **implementation.java** element and the
3055 **JavaImplementation** type contained in **sca-implementation-java.xsd**.

```
3056  
3057       <?xml version="1.0" encoding="UTF-8"?>  
3058       <schema xmlns="http://www.w3.org/2001/XMLSchema"  
3059               targetNamespace="http://docs.oasis-  
3060       open.org/ns/opencsa/sca/200712"  
3061               xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712">  
3062  
3063               <element name="implementation.java" type="sca:JavaImplementation"  
3064                       substitutionGroup="sca:implementation"/>  
3065               <complexType name="JavaImplementation">  
3066                <complexContent>  
3067                 <extension base="sca:Implementation">  
3068                  <attribute name="class" type="NCName"  
3069                  use="required"/>  
3070                </extension>  
3071                </complexContent>  
3072               </complexType>  
3073       </schema>
```

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

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

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

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

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

The `implementationType` has the following attributes:

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

11.3 Defining a Binding Type

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

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- binding type schema snippet -->
<!-- (c) Copyright SCA Collaboration 2006, 2007 -->
```

```

3118 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3119         targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3120         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3121         elementFormDefault="qualified">
3122
3123     ...
3124
3125     <element name="binding" type="sca:Binding" abstract="true"/>
3126     <complexType name="Binding">
3127         <attribute name="uri" type="anyURI" use="optional"/>
3128         <attribute name="name" type="NCName" use="optional"/>
3129         <attribute name="requires" type="sca:listOfQNames"
3130 use="optional"/>
3131         <attribute name="policySets" type="sca:listOfQNames"
3132 use="optional"/>
3133     </complexType>
3134
3135     ...
3136
3137 </schema>

```

3138 In the following snippet we show how the base definition is extended to support Web service
3139 binding. The snippet shows the definition of the **binding.ws** element and the
3140 **WebServiceBinding** type contained in **sca-binding-webservice.xsd**.

```

3141
3142 <?xml version="1.0" encoding="UTF-8"?>
3143 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3144         targetNamespace="http://docs.oasis-
3145 open.org/ns/opencsa/sca/200712"
3146         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712">
3147
3148     <element name="binding.ws" type="sca:WebServiceBinding"
3149         substitutionGroup="sca:binding"/>
3150     <complexType name="WebServiceBinding">
3151         <complexContent>
3152             <extension base="sca:Binding">
3153                 <attribute name="port" type="anyURI" use="required"/>
3154             </extension>
3155         </complexContent>
3156     </complexType>
3157 </schema>

```

3158 In the following snippet we show an example of how the base definition can be extended by other
3159 specifications to support a new binding not defined in the SCA specifications. The snippet shows
3160 the definition of the **my-binding-extension** element and the **my-binding-extension-type** type.

```

3161 <?xml version="1.0" encoding="UTF-8"?>
3162 <schema xmlns="http://www.w3.org/2001/XMLSchema"

```

```

3163         targetNamespace="http://www.example.org/myextension"
3164         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3165         xmlns:tns="http://www.example.org/myextension">
3166
3167     <element name="my-binding-extension" type="tns:my-binding-extension-
3168     type"
3169         substitutionGroup="sca:binding"/>
3170     <complexType name="my-binding-extension-type">
3171         <complexContent>
3172             <extension base="sca:Binding">
3173                 ...
3174             </extension>
3175         </complexContent>
3176     </complexType>
3177 </schema>
3178

```

3179 In addition to the definition for the new binding instance element, there needs to be an associated
3180 bindingType element which provides metadata about the new binding type. The pseudo schema
3181 for the bindingType element is shown in the following snippet:

```

3182 <bindingType type="xs:QName"
3183     alwaysProvides="list of intent QNames"?
3184     mayProvide = "list of intent QNames"?/>
3185

```

3186 The binding type has the following attributes:

- 3187 • **type (required)** – the type of the binding to which this bindingType element applies.
3188 This is intended to be the QName of the binding element for the binding type, such as
3189 "sca:binding.ws"
- 3190 • **alwaysProvides (optional)** – a set of intents which the binding type always provides.
3191 See [the Policy Framework specification \[10\]](#) for details.
- 3192 • **mayProvide (optional)** – a set of intents which the binding type may provide. See [the](#)
3193 [Policy Framework specification \[10\]](#) for details.

3194 12 Packaging and Deployment

3195 12.1 Domains

3196 An **SCA Domain** represents a complete runtime configuration, potentially distributed over a series
3197 of interconnected runtime nodes.

3198 A single SCA domain defines the boundary of visibility for all SCA mechanisms. For example, SCA
3199 wires can only be used to connect components within a single SCA domain. Connections to
3200 services outside the domain must use binding specific mechanisms for addressing services (such
3201 as WSDL endpoint URIs). Also, SCA mechanisms such as intents and policySets can only be used
3202 in the context of a single domain. In general, external clients of a service that is developed and
3203 deployed using SCA should not be able to tell that SCA was used to implement the service – it is
3204 an implementation detail.

3205 The size and configuration of an SCA Domain is not constrained by the SCA Assembly specification
3206 and is expected to be highly variable. An SCA Domain typically represents an area of business
3207 functionality controlled by a single organization. For example, an SCA Domain may be the whole
3208 of a business, or it may be a department within a business.

3209 As an example, for the accounts department in a business, the SCA Domain might cover all
3210 finance-related functions, and it might contain a series of composites dealing with specific areas of
3211 accounting, with one for Customer accounts and another dealing with Accounts Payable.

3212 An SCA domain has the following:

- 3213 • A virtual domain-level composite whose components are deployed and running
- 3214 • A set of *installed contributions* that contain implementations, interfaces and other artifacts
3215 necessary to execute components
- 3216 • A set of logical services for manipulating the set of contributions and the virtual domain-
3217 level composite.

3218 The information associated with an SCA domain can be stored in many ways, including but not
3219 limited to a specific filesystem structure or a repository.

3220 12.2 Contributions

3221 An SCA domain may require a large number of different artifacts in order to work. These artifacts
3222 include artifacts defined by SCA and other artifacts such as object code files and interface
3223 definition files. The SCA-defined artifact types are all XML documents. The root elements of the
3224 different SCA definition documents are: composite, componentType, constrainingType and
3225 definitions. XML artifacts that are not defined by SCA but which may be needed by an SCA
3226 domain include XML Schema documents, WSDL documents, and BPEL documents. SCA
3227 constructs, like other XML-defined constructs, use XML qualified names for their identity (i.e.
3228 namespace + local name).

3229 Non-XML artifacts are also required within an SCA domain. The most obvious examples of such
3230 non-XML artifacts are Java, C++ and other programming language files necessary for component
3231 implementations. Since SCA is extensible, other XML and non-XML artifacts may also be required.

3232 SCA defines an interoperable packaging format for contributions (ZIP), as specified below. This
3233 format is not the only packaging format that an SCA runtime can use. SCA allows many different
3234 packaging formats, but requires that the ZIP format be supported. When using the ZIP format for
3235 deploying a contribution, this specification does not specify whether that format is retained after
3236 deployment. For example, a Java EE based SCA runtime may convert the ZIP package to an EAR
3237 package. SCA expects certain characteristics of any packaging:

- 3238 • It must be possible to present the artifacts of the packaging to SCA as a hierarchy of
3239 resources based off of a single root

3240 • A directory resource should exist at the root of the hierarchy named META-INF
 3241 • A document should exist directly under the META-INF directory named sca-
 3242 contribution.xml which lists the SCA Composites within the contribution that are runnable.
 3243
 3244 The same document also optionally lists namespaces of constructs that are defined within
 3245 the contribution and which may be used by other contributions
 3246 Optionally, additional elements may exist that list the namespaces of constructs that are
 3247 needed by the contribution and which must be found elsewhere, for example in other
 3248 contributions. These optional elements may not be physically present in the packaging,
 3249 but may be generated based on the definitions and references that are present, or they
 3250 may not exist at all if there are no unresolved references.
 3251
 3252 See the section "SCA Contribution Metadata Document" for details of the format of this
 3253 file.
 3254
 3255 To illustrate that a variety of packaging formats can be used with SCA, the following are examples
 3256 of formats that might be used to package SCA artifacts and metadata (as well as other artifacts)
 3257 as a contribution:
 3258 • A filesystem directory
 3259 • An OSGi bundle
 3260 • A compressed directory (zip, gzip, etc)
 3261 • A JAR file (or its variants – WAR, EAR, etc)
 3262
 3263 Contributions do not contain other contributions. If the packaging format is a JAR file that
 3264 contains other JAR files (or any similar nesting of other technologies), the internal files are not
 3265 treated as separate SCA contributions. It is up to the implementation to determine whether the
 3266 internal JAR file should be represented as a single artifact in the contribution hierarchy or whether
 3267 all of the contents should be represented as separate artifacts.
 3268 A goal of SCA's approach to deployment is that the contents of a contribution should not need to
 3269 be modified in order to install and use the contents of the contribution in a domain.

3269 12.2.1 SCA Artifact Resolution

3270 Contributions may be self-contained, in that all of the artifacts necessary to run the contents of
 3271 the contribution are found within the contribution itself. However, it may also be the case that the
 3272 contents of the contribution make one or many references to artifacts that are not contained
 3273 within the contribution. These references may be to SCA artifacts or they may be to other
 3274 artifacts such as WSDL files, XSD files or to code artifacts such as Java class files and BPEL scripts.
 3275 A contribution may use some artifact-related or packaging-related means to resolve artifact
 3276 references. Examples of such mechanisms include:
 3277 • wsdlLocation and schemaLocation attributes in references to WSDL and XSD schema
 3278 artifacts respectively
 3279 • OSGi bundle mechanisms for resolving Java class and related resource dependencies
 3280 Where present, these mechanisms must be used to resolve artifact dependencies.
 3281 SCA also provides an artifact resolution mechanism. The SCA artifact resolution mechanisms are
 3282 used either where no other mechanisms are available, or in cases where the mechanisms used by
 3283 the various contributions in the same SCA Domain are different. An example of the latter case is
 3284 where an OSGi Bundle is used for one contribution but where a second contribution used by the
 3285 first one is not implemented using OSGi - eg the second contribution is a mainframe COBOL
 3286 service whose interfaces are declared using WSDL which must be accessed by the first
 3287 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.

12.2.2 SCA Contribution Metadata Document

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

The format of the document is:

```
<?xml version="1.0" encoding="ASCII"?>
<!-- sca-contribution pseudo-schema -->
<contribution xmlns=http://docs.oasis-open.org/ns/opencsa/sca/200712>

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

</contribution>
```

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

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

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

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

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

Technologies that use naming schemes other than QNames must use a different export

Deleted: 23 September

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

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

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

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

- **location (optional)** – a URI to resolve the definitions for this import. SCA makes no specific requirements for the form of this URI, nor the means by which it is resolved. It may point to another contribution (through its URI) or it may point to some location entirely outside the SCA Domain.

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

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

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

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

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

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

12.2.3 Contribution Packaging using ZIP

SCA allows many different packaging formats that SCA runtimes can support, but SCA requires that all runtimes support the ZIP packaging format for contributions. This format allows that

Deleted: 23 September

metadata specified by the section 'SCA Contribution Metadata Document' be present. Specifically, it may contain a top-level "META-INF" directory and a "META-INF/sca-contribution.xml" file and there may also be an optional "META-INF/sca-contribution-generated.xml" file in the package. SCA defined artifacts as well as non-SCA defined artifacts such as object files, WSDL definition, Java classes may be present anywhere in the ZIP archive,

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

12.3 Installed Contribution

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

An installed contribution is made up of the following things:

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

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

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

12.3.1 Installed Artifact URIs

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

12.4 Operations for Contributions

SCA Domains provide the following conceptual functionality associated with contributions (meaning the function may not be represented as addressable services and also meaning that

3436 equivalent functionality may be provided in other ways). The functionality is optional meaning that
3437 some SCA runtimes may choose not to provide that functionality in any way:

3438 12.4.1 install Contribution & update Contribution

3439 Creates or updates an installed contribution with a supplied root contribution, and installed at a
3440 supplied base URI. A supplied dependent contribution list specifies the contributions that should
3441 be used to resolve the dependencies of the root contribution and other dependent contributions.
3442 These override any dependent contributions explicitly listed via the location attribute in the import
3443 statements of the contribution.
3444

3445 SCA follows the simplifying assumption that the use of a contribution for resolving anything also
3446 means that all other exported artifacts can be used from that contribution. Because of this, the
3447 dependent contribution list is just a list of installed contribution URIs. There is no need to specify
3448 what is being used from each one.
3449

3450 Each dependent contribution is also an installed contribution, with its own dependent
3451 contributions. By default these dependent contributions of the dependent contributions (which we
3452 will call *indirect dependent contributions*) are included as dependent contributions of the installed
3453 contribution. However, if a contribution in the dependent contribution list exports any conflicting
3454 definitions with an indirect dependent contribution, then the indirect dependent contribution is not
3455 included (i.e. the explicit list overrides the default inclusion of indirect dependent contributions).
3456 Also, if there is ever a conflict between two indirect dependent contributions, then the conflict
3457 must be resolved by an explicit entry in the dependent contribution list.

3458 Note that in many cases, the dependent contribution list can be generated. In particular, if a
3459 domain is careful to avoid creating duplicate definitions for the same qualified name, then it is
3460 easy for this list to be generated by tooling.

3461 12.4.2 add Deployment Composite & update Deployment Composite

3462 Adds or updates a deployment composite using a supplied composite ("composite by value" – a
3463 data structure, not an existing resource in the domain) to the contribution identified by a supplied
3464 contribution URI. The added or updated deployment composite is given a relative URI that
3465 matches the @name attribute of the composite, with a ".composite" suffix. Since all composites
3466 must run within the context of a installed contribution (any component implementations or other
3467 definitions are resolved within that contribution), this functionality makes it possible for the
3468 deployer to create a composite with final configuration and wiring decisions and add it to an
3469 installed contribution without having to modify the contents of the root contribution.

3470 Also, in some use cases, a contribution may include only implementation code (e.g. PHP scripts).
3471 It should then be possible for those to be given component names by a (possibly generated)
3472 composite that is added into the installed contribution, without having to modify the packaging.

3473 12.4.3 remove Contribution

3474 Removes the deployed contribution identified by a supplied contribution URI.

3475

3476 12.5 Use of Existing (non-SCA) Mechanisms for Resolving Artifacts

3477

3478 For certain types of artifact, there are existing and commonly used mechanisms for referencing a
3479 specific concrete location where the artifact can be resolved.

3480 Examples of these mechanisms include:

- 3481 • For WSDL files, the **@wsdlLocation** attribute is a hint that has a URI value pointing to the
3482 place holding the WSDL itself.

3483 • For XSDs, the **@schemaLocation** attribute is a hint which matches the namespace to a
3484 URI where the XSD is found.

3485 **Note:** In neither of these cases is the runtime obliged to use the location hint and the URI does
3486 not have to be dereferenced.

3487 SCA permits the use of these mechanisms. Where present, these mechanisms take precedence
3488 over the SCA mechanisms. However, use of these mechanisms is discouraged because tying
3489 assemblies to addresses in this way makes the assemblies less flexible and prone to errors when
3490 changes are made to the overall SCA Domain.

3491 **Note:** If one of these mechanisms is present, but there is a failure to find the resource indicated
3492 when using the mechanism (eg the URI is incorrect or invalid, say) the SCA runtime MUST raise
3493 an error and MUST NOT attempt to use SCA resolution mechanisms as an alternative.

3494

3495 **12.6 Domain-Level Composite**

3496 The domain-level composite is a virtual composite, in that it is not defined by a composite
3497 definition document. Rather, it is built up and modified through operations on the domain.
3498 However, in other respects it is very much like a composite, since it contains components, wires,
3499 services and references.

3500

3501 The value of @autowire for the logical domain composite MUST be autowire="false".

Comment [mbgl14]: Issue
42

3502

3503 For components at the Domain level, with References for which @autowire="true" applies, the
3504 behaviour of the SCA runtime for a given Domain MUST take ONE of the 3 following forms:

Comment [mbgl15]: Issue
40

- 3505 1. The SCA runtime MAY disallow deployment of any components with autowire References.
3506 In this case, the SCA runtime MUST generate an exception at the point where the
3507 component is deployed.
- 3508 2. The SCA runtime MAY evaluate the target(s) for the reference at the time that the
3509 component is deployed and not update those targets when later deployment actions occur.
- 3510 3. The SCA runtime MAY re-evaluate the target(s) for the reference dynamically as later
3511 deployment actions occur resulting in updated reference targets which match the new
3512 Domain configuration. How the new configuration of the reference takes place is described
3513 by the relevant client and implementation specifications.

3514

3515 The abstract domain-level functionality for modifying the domain-level composite is as follows,
3516 although a runtime may supply equivalent functionality in a different form:

3517 **12.6.1 add To Domain-Level Composite**

3518 This functionality adds the composite identified by a supplied URI to the Domain Level Composite.
3519 The supplied composite URI must refer to a composite within a installed contribution. The
3520 composite's installed contribution determines how the composite's artifacts are resolved (directly
3521 and indirectly). The supplied composite is added to the domain composite with semantics that
3522 correspond to the domain-level composite having an <include> statement that references the
3523 supplied composite. All of the composite's components become *top-level* components and the
3524 services become externally visible services (eg. they would be present in a WSDL description of
3525 the domain).

3526 **12.6.2 remove From Domain-Level Composite**

3527 Removes from the Domain Level composite the elements corresponding to the composite
3528 identified by a supplied composite URI. This means that the removal of the components, wires,

Deleted: 23 September

3529 services and references originally added to the domain level composite by the identified
3530 composite.

3531 12.6.3 get Domain-Level Composite

3532 Returns a <composite> definition that has an <include> line for each composite that had been
3533 added to the domain level composite. It is important to note that, in dereferencing the included
3534 composites, any referenced artifacts must be resolved in terms of that installed composite.

3535 12.6.4 get QName Definition

3536 In order to make sense of the domain-level composite (as returned by get Domain-Level
3537 Composite), it must be possible to get the definitions for named artifacts in the included
3538 composites. This functionality takes the supplied URI of an installed contribution (which provides
3539 the context), a supplied qualified name of a definition to look up, and a supplied symbol space (as
3540 a QName, eg wsdl:PortType). The result is a single definition, in whatever form is appropriate for
3541 that definition type.

3542 Note that this, like all the other domain-level operations, is a conceptual operation. Its capabilities
3543 should exist in some form, but not necessarily as a service operation with exactly this signature.

3544
3545
3546

13 Conformance

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

A. Pseudo Schema

A.1 ComponentType

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

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

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

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

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


3587
3588 </componentType>
3589

3590 A.2 Composite

```
3591   <?xml version="1.0" encoding="ASCII"?>
3592   <!-- Composite schema snippet -->
3593   <composite      xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3594                 targetNamespace="xs:anyURI"
3595                 name="xs:NCName" local="xs:boolean"?
3596                 autowire="xs:boolean"? constrainingType="QName"?
3597                 requires="list of xs:QName"? policySets="list of
3598 xs:QName"?>
3599
3600       <include name="xs:QName"/>*
3601
3602       <service name="xs:NCName" promote="xs:anyURI"
3603             requires="list of xs:QName"? policySets="list of xs:QName"?>*
3604         <interface ... />?
3605         <binding uri="xs:anyURI"? name="xs:NCName"?
3606             requires="list of xs:QName"? policySets="list of
3607 xs:QName"?/>*
3608         <callback?>
3609             <binding uri="xs:anyURI"? name="xs:NCName"?
3610                 requires="list of xs:QName"?
3611                 policySets="list of xs:QName"?/>+
3612         </callback>
3613     </service>
3614
3615     <reference name="xs:NCName" target="list of xs:anyURI"?
3616             promote="list of xs:anyURI" wiredByImpl="xs:boolean"?
3617             multiplicity="0..1 or 1..1 or 0..n or 1..n"?
3618             requires="list of xs:QName"? policySets="list of xs:QName"?>*
3619         <interface ... />?
3620         <binding uri="xs:anyURI"? name="xs:NCName"?
3621             requires="list of xs:QName"? policySets="list of
3622 xs:QName"?/>*
3623         <callback?>
3624             <binding uri="xs:anyURI"? name="xs:NCName"?
3625                 requires="list of xs:QName"?
3626                 policySets="list of xs:QName"?/>+
3627         </callback>
3628     </reference>
3629
```

```

3630 <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
3631     many="xs:boolean"? mustSupply="xs:boolean"?>*
3632     default-property-value?
3633 </property>
3634
3635 <component name="xs:NCName" autowire="xs:boolean"?
3636     requires="list of xs:QName"? policySets="list of xs:QName"?>*
3637     <implementation ... />?
3638     <service name="xs:NCName" requires="list of xs:QName"?
3639         policySets="list of xs:QName"?>*
3640         <interface ... />?
3641         <binding uri="xs:anyURI"? name="xs:NCName"?
3642             requires="list of xs:QName"?
3643             policySets="list of xs:QName"?/>*
3644         <callback>?
3645             <binding uri="xs:anyURI"? name="xs:NCName"?
3646                 requires="list of xs:QName"?
3647                 policySets="list of xs:QName"?/>+
3648         </callback>
3649     </service>
3650 <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
3651     source="xs:string"? file="xs:anyURI"? value="xs:string"?>*
3652     [<value>+ | xs:any+]?
3653 </property>
3654 <reference name="xs:NCName" target="list of xs:anyURI"?
3655     autowire="xs:boolean"? wiredByImpl="xs:boolean"?
3656     requires="list of xs:QName"? policySets="list of xs:QName"?
3657     multiplicity="0..1 or 1..1 or 0..n or 1..n"?/>*
3658     <interface ... />?
3659     <binding uri="xs:anyURI"? name="xs:NCName"?
3660         requires="list of xs:QName"?
3661         policySets="list of xs:QName"?/>*
3662     <callback>?
3663         <binding uri="xs:anyURI"? name="xs:NCName"?
3664             requires="list of xs:QName"?
3665             policySets="list of xs:QName"?/>+
3666     </callback>
3667 </reference>
3668 </component>
3669
3670 <wire source="xs:anyURI" target="xs:anyURI" />*
3671
3672 </composite>

```

B. XML Schemas

B.1 sca.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2006 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
  xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712">

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

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

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

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

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

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

Comment [mbgl16]: Issue 28

B.2 sca-core.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2006, 2007 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
  xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
  elementFormDefault="qualified">

  <element name="componentType" type="sca:ComponentType"/>
  <complexType name="ComponentType">
```

Deleted: 23 September

```

3712     <sequence>
3713         <element ref="sca:implementation" minOccurs="0" maxOccurs="1"/>
3714         <choice minOccurs="0" maxOccurs="unbounded">
3715             <element name="service" type="sca:ComponentService" />
3716             <element name="reference" type="sca:ComponentReference" />
3717             <element name="property" type="sca:Property" />
3718         </choice>
3719         <any namespace="##other" processContents="lax" minOccurs="0"
3720             maxOccurs="unbounded" />
3721     </sequence>
3722     <attribute name="constrainingType" type="QName" use="optional"/>
3723     <anyAttribute namespace="##other" processContents="lax"/>
3724 </complexType>
3725
3726 <element name="composite" type="sca:Composite"/>
3727 <complexType name="Composite">
3728     <sequence>
3729         <element name="include" type="anyURI" minOccurs="0"
3730             maxOccurs="unbounded" />
3731         <choice minOccurs="0" maxOccurs="unbounded">
3732             <element name="service" type="sca:Service"/>
3733             <element name="property" type="sca:Property"/>
3734             <element name="component" type="sca:Component"/>
3735             <element name="reference" type="sca:Reference"/>
3736             <element name="wire" type="sca:Wire"/>
3737         </choice>
3738         <any namespace="##other" processContents="lax" minOccurs="0"
3739             maxOccurs="unbounded" />
3740     </sequence>
3741     <attribute name="name" type="NCName" use="required"/>
3742     <attribute name="targetNamespace" type="anyURI" use="required"/>
3743     <attribute name="local" type="boolean" use="optional"
3744 default="false"/>
3745     <attribute name="autowire" type="boolean" use="optional"
3746 default="false"/>
3747     <attribute name="constrainingType" type="QName" use="optional"/>
3748     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3749     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3750     <anyAttribute namespace="##other" processContents="lax"/>
3751 </complexType>
3752
3753 <complexType name="Service">
3754     <sequence>

```

```

3755     <element ref="sca:interface" minOccurs="0" maxOccurs="1" />
3756     <element name="operation" type="sca:Operation" minOccurs="0"
3757         maxOccurs="unbounded" />
3758     <choice minOccurs="0" maxOccurs="unbounded">
3759         <element ref="sca:binding" />
3760         <any namespace="##other" processContents="lax"
3761             minOccurs="0" maxOccurs="unbounded" />
3762     </choice>
3763     <element ref="sca:callback" minOccurs="0" maxOccurs="1" />
3764     <any namespace="##other" processContents="lax" minOccurs="0"
3765         maxOccurs="unbounded" />
3766 </sequence>
3767 <attribute name="name" type="NCName" use="required" />
3768 <attribute name="promote" type="anyURI" use="required" />
3769 <attribute name="requires" type="sca:listOfQNames" use="optional" />
3770 <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3771 <anyAttribute namespace="##other" processContents="lax" />
3772 </complexType>
3773
3774 <element name="interface" type="sca:Interface" abstract="true" />
3775 |-----<complexType name="Interface" abstract="true">
3776     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3777     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3778 </complexType>
3779
3780 <complexType name="Reference">
3781     <sequence>
3782         <element ref="sca:interface" minOccurs="0" maxOccurs="1" />
3783         <element name="operation" type="sca:Operation" minOccurs="0"
3784             maxOccurs="unbounded" />
3785         <choice minOccurs="0" maxOccurs="unbounded">
3786             <element ref="sca:binding" />
3787             <any namespace="##other" processContents="lax" />
3788         </choice>
3789         <element ref="sca:callback" minOccurs="0" maxOccurs="1" />
3790         <any namespace="##other" processContents="lax" minOccurs="0"
3791             maxOccurs="unbounded" />
3792     </sequence>
3793     <attribute name="name" type="NCName" use="required" />
3794     <attribute name="target" type="sca:listOfAnyURIs" use="optional"/>
3795     <attribute name="wiredByImpl" type="boolean" use="optional"
3796         default="false"/>
3797     <attribute name="multiplicity" type="sca:Multiplicity"
3798         use="optional" default="1..1" />

```

Comment [mbgl17]: Issue
39

Deleted: 23 September

```

3799     <attribute name="promote" type="sca:listOfAnyURIs" use="required" />
3800     <attribute name="requires" type="sca:listOfQNames" use="optional" />
3801     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3802     <anyAttribute namespace="##other" processContents="lax" />
3803 </complexType>
3804
3805 <complexType name="SCAPropertyBase" mixed="true">
3806     <!-- mixed="true" to handle simple type -->
3807     <sequence>
3808         <choice minOccurs="0">
3809             <element name="value" minOccurs="1" maxOccurs="unbounded"
3810                 type="anyType"/>
3811             <any namespace="##any" processContents="lax" minOccurs="1"
3812                 maxOccurs="unbounded" />
3813             <!-- NOT an extension point; This xsd:any exists
3814                 to accept the element-based or complex type
3815                 property i.e. no element-based extension point
3816                 under "sca:property" -->
3817         </choice>
3818     </sequence>
3819 </complexType>
3820
3821 <!-- complex type for sca:property declaration -->
3822 <complexType name="Property" mixed="true">
3823     <complexContent>
3824         <extension base="sca:SCAPropertyBase">
3825             <!-- extension defines the place to hold default value -->
3826             <attribute name="name" type="NCName" use="required"/>
3827             <attribute name="value" type="xs:string" use="optional"/>
3828             <attribute name="type" type="QName" use="optional"/>
3829             <attribute name="element" type="QName" use="optional"/>
3830             <attribute name="many" type="boolean" default="false"
3831                 use="optional"/>
3832             <attribute name="mustSupply" type="boolean" default="false"
3833                 use="optional"/>
3834             <anyAttribute namespace="##other" processContents="lax"/>
3835             <!-- an extension point ; attribute-based only -->
3836         </extension>
3837     </complexContent>
3838 </complexType>
3839
3840 <complexType name="PropertyValue" mixed="true">
3841     <complexContent>

```

```

3842     <extension base="sca:SCAPropertyBase">
3843         <attribute name="name" type="NCName" use="required"/>
3844         <attribute name="value" type="xs:string" use="optional"/>
3845         <attribute name="type" type="QName" use="optional"/>
3846         <attribute name="element" type="QName" use="optional"/>
3847         <attribute name="many" type="boolean" default="false"
3848             use="optional"/>
3849         <attribute name="source" type="string" use="optional"/>
3850         <attribute name="file" type="anyURI" use="optional"/>
3851         <anyAttribute namespace="##other" processContents="lax"/>
3852         <!-- an extension point ; attribute-based only -->
3853     </extension>
3854 </complexContent>
3855 </complexType>
3856
3857 <element name="binding" type="sca:Binding" abstract="true"/>
3858 <complexType name="Binding" abstract="true">
3859     <sequence>
3860         <element name="operation" type="sca:Operation" minOccurs="0"
3861             maxOccurs="unbounded" />
3862     </sequence>
3863     <attribute name="uri" type="anyURI" use="optional"/>
3864     <attribute name="name" type="NCName" use="optional"/>
3865     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3866     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3867 </complexType>
3868
3869 <element name="bindingType" type="sca:BindingType"/>
3870 <complexType name="BindingType">
3871     <sequence minOccurs="0" maxOccurs="unbounded">
3872         <any namespace="##other" processContents="lax" />
3873     </sequence>
3874     <attribute name="type" type="QName" use="required"/>
3875     <attribute name="alwaysProvides" type="sca:listOfQNames"
3876 use="optional"/>
3877     <attribute name="mayProvide" type="sca:listOfQNames" use="optional"/>
3878     <anyAttribute namespace="##other" processContents="lax"/>
3879 </complexType>
3880
3881 <element name="callback" type="sca:Callback"/>
3882 <complexType name="Callback">
3883     <choice minOccurs="0" maxOccurs="unbounded">
3884         <element ref="sca:binding"/>

```

```

3885         <any namespace="##other" processContents="lax"/>
3886     </choice>
3887     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3888     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3889     <anyAttribute namespace="##other" processContents="lax"/>
3890 </complexType>
3891
3892 <complexType name="Component">
3893     <sequence>
3894         <element ref="sca:implementation" minOccurs="0" maxOccurs="1"/>
3895         <choice minOccurs="0" maxOccurs="unbounded">
3896             <element name="service" type="sca:ComponentService"/>
3897             <element name="reference" type="sca:ComponentReference"/>
3898             <element name="property" type="sca:PropertyValue" />
3899         </choice>
3900         <any namespace="##other" processContents="lax" minOccurs="0"
3901             maxOccurs="unbounded" />
3902     </sequence>
3903     <attribute name="name" type="NCName" use="required"/>
3904     <attribute name="autowire" type="boolean" use="optional" />
3905     <attribute name="constrainingType" type="QName" use="optional"/>
3906     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3907     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3908     <anyAttribute namespace="##other" processContents="lax"/>
3909 </complexType>
3910
3911 <complexType name="ComponentService">
3912     <complexContent>
3913         <restriction base="sca:Service">
3914             <sequence>
3915                 <element ref="sca:interface" minOccurs="0"
3916 maxOccurs="1"/>
3917                 <element name="operation" type="sca:Operation"
3918 minOccurs="0"
3919                 maxOccurs="unbounded" />
3920                 <choice minOccurs="0" maxOccurs="unbounded">
3921                     <element ref="sca:binding"/>
3922                     <any namespace="##other" processContents="lax"
3923                         minOccurs="0" maxOccurs="unbounded"/>
3924                 </choice>
3925                 <element ref="sca:callback" minOccurs="0"
3926 maxOccurs="1"/>
3927                 <any namespace="##other" processContents="lax"
3928 minOccurs="0"

```



```

3929             maxOccurs="unbounded" />
3930         </sequence>
3931         <attribute name="name" type="NCName" use="required"/>
3932         <attribute name="requires" type="sca:listOfQNames"
3933             use="optional"/>
3934         <attribute name="policySets" type="sca:listOfQNames"
3935             use="optional"/>
3936         <anyAttribute namespace="##other" processContents="lax"/>
3937     </restriction>
3938 </complexContent>
3939 </complexType>
3940
3941 <complexType name="ComponentReference">
3942     <complexContent>
3943         <restriction base="sca:Reference">
3944             <sequence>
3945                 <element ref="sca:interface" minOccurs="0"
3946 maxOccurs="1" />
3947                 <element name="operation" type="sca:Operation"
3948 minOccurs="0"
3949                 maxOccurs="unbounded" />
3950                 <choice minOccurs="0" maxOccurs="unbounded">
3951                     <element ref="sca:binding" />
3952                     <any namespace="##other" processContents="lax"
3953 />
3954                 </choice>
3955                 <element ref="sca:callback" minOccurs="0"
3956 maxOccurs="1" />
3957                 <any namespace="##other" processContents="lax"
3958 minOccurs="0"
3959                 maxOccurs="unbounded" />
3960             </sequence>
3961             <attribute name="name" type="NCName" use="required" />
3962             <attribute name="autowire" type="boolean" use="optional" />
3963             <attribute name="wiredByImpl" type="boolean" use="optional"
3964                 default="false"/>
3965             <attribute name="target" type="sca:listOfAnyURIs"
3966 use="optional"/>
3967             <attribute name="multiplicity" type="sca:Multiplicity"
3968                 use="optional" default="1..1" />
3969             <attribute name="requires" type="sca:listOfQNames"
3970 use="optional"/>
3971             <attribute name="policySets" type="sca:listOfQNames"
3972                 use="optional"/>
3973             <anyAttribute namespace="##other" processContents="lax" />

```

```

3974         </restriction>
3975     </complexContent>
3976 </complexType>
3977
3978 <element name="implementation" type="sca:Implementation"
3979     abstract="true" />
3980 <complexType name="Implementation" abstract="true">
3981     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3982     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3983 </complexType>
3984
3985 <element name="implementationType" type="sca:ImplementationType"/>
3986 <complexType name="ImplementationType">
3987     <sequence minOccurs="0" maxOccurs="unbounded">
3988         <any namespace="##other" processContents="lax" />
3989     </sequence>
3990     <attribute name="type" type="QName" use="required"/>
3991     <attribute name="alwaysProvides" type="sca:listOfQNames"
3992 use="optional"/>
3993     <attribute name="mayProvide" type="sca:listOfQNames" use="optional"/>
3994     <anyAttribute namespace="##other" processContents="lax"/>
3995 </complexType>
3996
3997 <complexType name="Wire">
3998     <sequence>
3999         <any namespace="##other" processContents="lax" minOccurs="0"
4000             maxOccurs="unbounded" />
4001     </sequence>
4002     <attribute name="source" type="anyURI" use="required"/>
4003     <attribute name="target" type="anyURI" use="required"/>
4004     <anyAttribute namespace="##other" processContents="lax"/>
4005 </complexType>
4006
4007 <element name="include" type="sca:Include"/>
4008 <complexType name="Include">
4009     <attribute name="name" type="QName"/>
4010     <anyAttribute namespace="##other" processContents="lax"/>
4011 </complexType>
4012
4013 <complexType name="Operation">
4014     <attribute name="name" type="NCName" use="required"/>
4015     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
4016     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>

```

```

4017     <anyAttribute namespace="##other" processContents="lax" />
4018 </complexType>
4019
4020 <element name="constrainingType" type="sca:ConstrainingType" />
4021 <complexType name="ConstrainingType">
4022     <sequence>
4023         <choice minOccurs="0" maxOccurs="unbounded">
4024             <element name="service" type="sca:ComponentService" />
4025             <element name="reference" type="sca:ComponentReference" />
4026             <element name="property" type="sca:Property" />
4027         </choice>
4028         <any namespace="##other" processContents="lax" minOccurs="0"
4029             maxOccurs="unbounded" />
4030     </sequence>
4031     <attribute name="name" type="NCName" use="required" />
4032     <attribute name="targetNamespace" type="anyURI" />
4033     <attribute name="requires" type="sca:listOfQNames" use="optional" />
4034     <anyAttribute namespace="##other" processContents="lax" />
4035 </complexType>
4036
4037
4038 <simpleType name="Multiplicity">
4039     <restriction base="string">
4040         <enumeration value="0..1" />
4041         <enumeration value="1..1" />
4042         <enumeration value="0..n" />
4043         <enumeration value="1..n" />
4044     </restriction>
4045 </simpleType>
4046
4047 <simpleType name="OverrideOptions">
4048     <restriction base="string">
4049         <enumeration value="no" />
4050         <enumeration value="may" />
4051         <enumeration value="must" />
4052     </restriction>
4053 </simpleType>
4054
4055 <!-- Global attribute definition for @requires to permit use of intents
4056     within WSDL documents -->
4057 <attribute name="requires" type="sca:listOfQNames" />
4058
4059 <!-- Global attribute defintion for @endsConversation to mark operations

```

```

4060         as ending a conversation -->
4061     <attribute name="endsConversation" type="boolean" default="false"/>
4062
4063     <simpleType name="listOfQNames">
4064         <list itemType="QName"/>
4065     </simpleType>
4066
4067     <simpleType name="listOfAnyURIs">
4068         <list itemType="anyURI"/>
4069     </simpleType>
4070
4071 </schema>

```

4072 B.3 sca-binding-sca.xsd

```

4073
4074 <?xml version="1.0" encoding="UTF-8"?>
4075 <!-- (c) Copyright SCA Collaboration 2006, 2007 -->
4076 <schema xmlns="http://www.w3.org/2001/XMLSchema"
4077     targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4078     xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4079     elementFormDefault="qualified">
4080
4081     <include schemaLocation="sca-core.xsd"/>
4082
4083     <element name="binding.sca" type="sca:SCABinding"
4084         substitutionGroup="sca:binding"/>
4085     <complexType name="SCABinding">
4086         <complexContent>
4087             <extension base="sca:Binding">
4088                 <sequence>
4089                     <element name="operation" type="sca:Operation"
4090 minOccurs="0"
4091                         maxOccurs="unbounded" />
4092                 </sequence>
4093                 <attribute name="uri" type="anyURI" use="optional"/>
4094                 <attribute name="name" type="QName" use="optional"/>
4095                 <attribute name="requires" type="sca:listOfQNames"
4096                     use="optional"/>
4097                 <attribute name="policySets" type="sca:listOfQNames"
4098                     use="optional"/>
4099                 <anyAttribute namespace="##other" processContents="lax"/>
4100             </extension>
4101         </complexContent>

```

4102 </complexType>
4103 </schema>
4104

4105 B.4 sca-interface-java.xsd

4106
4107 <?xml version="1.0" encoding="UTF-8"?>
4108 <!-- (c) Copyright SCA Collaboration 2006 -->
4109 <schema xmlns="http://www.w3.org/2001/XMLSchema"
4110 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4111 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4112 elementFormDefault="qualified">
4113
4114 <include schemaLocation="sca-core.xsd"/>
4115
4116 <element name="interface.java" type="sca:JavaInterface"
4117 substitutionGroup="sca:interface"/>
4118 <complexType name="JavaInterface">
4119 <complexContent>
4120 <extension base="sca:Interface">
4121 <sequence>
4122 <any namespace="##other" processContents="lax"
4123 minOccurs="0" maxOccurs="unbounded"/>
4124 </sequence>
4125 <attribute name="interface" type="NCName" use="required"/>
4126 <attribute name="callbackInterface" type="NCName"
4127 use="optional"/>
4128 <anyAttribute namespace="##other" processContents="lax"/>
4129 </extension>
4130 </complexContent>
4131 </complexType>
4132 </schema>
4133

4134 B.5 sca-interface-wsdl.xsd

4135
4136 <?xml version="1.0" encoding="UTF-8"?>
4137 <!-- (c) Copyright SCA Collaboration 2006 -->
4138 <schema xmlns="http://www.w3.org/2001/XMLSchema"
4139 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4140 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4141 elementFormDefault="qualified">
4142
4143 <include schemaLocation="sca-core.xsd"/>

```

4144
4145     <element name="interface.wsdl" type="sca:WSDLPortType"
4146           substitutionGroup="sca:interface" />
4147   <complexType name="WSDLPortType">
4148     <complexContent>
4149       <extension base="sca:Interface">
4150         <sequence>
4151           <any namespace="##other" processContents="lax"
4152 minOccurs="0"           maxOccurs="unbounded" />
4153         </sequence>
4154         <attribute name="interface" type="anyURI" use="required" />
4155         <attribute name="callbackInterface" type="anyURI"
4156 use="optional" />
4157         <anyAttribute namespace="##other" processContents="lax" />
4158       </extension>
4159     </complexContent>
4160   </complexType>
4161 </schema>
4162

```

4163 B.6 sca-implementation-java.xsd

```

4164
4165 <?xml version="1.0" encoding="UTF-8"?>
4166 <!-- (c) Copyright SCA Collaboration 2006 -->
4167 <schema xmlns="http://www.w3.org/2001/XMLSchema"
4168       targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4169       xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4170       elementFormDefault="qualified">
4171
4172   <include schemaLocation="sca-core.xsd" />
4173
4174   <element name="implementation.java" type="sca:JavaImplementation"
4175         substitutionGroup="sca:implementation" />
4176   <complexType name="JavaImplementation">
4177     <complexContent>
4178       <extension base="sca:Implementation">
4179         <sequence>
4180           <any namespace="##other" processContents="lax"
4181 minOccurs="0" maxOccurs="unbounded" />
4182         </sequence>
4183         <attribute name="class" type="NCName" use="required" />
4184         <attribute name="requires" type="sca:listOfQNames"
4185 use="optional" />
4186         <attribute name="policySets" type="sca:listOfQNames"

```

Deleted: 23 September

```

4187         use="optional"/>
4188     <anyAttribute namespace="##other" processContents="lax"/>
4189 </extension>
4190 </complexContent>
4191 </complexType>
4192 </schema>

```

4193 B.7 sca-implementation-composite.xsd

```

4194
4195 <?xml version="1.0" encoding="UTF-8"?>
4196 <!-- (c) Copyright SCA Collaboration 2006 -->
4197 <schema xmlns="http://www.w3.org/2001/XMLSchema"
4198     targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4199     xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4200     elementFormDefault="qualified">
4201
4202     <include schemaLocation="sca-core.xsd"/>
4203     <element name="implementation.composite" type="sca:SCAImplementation"
4204         substitutionGroup="sca:implementation"/>
4205     <complexType name="SCAImplementation">
4206         <complexContent>
4207             <extension base="sca:Implementation">
4208                 <sequence>
4209                     <any namespace="##other" processContents="lax"
4210 minOccurs="0"
4211                         maxOccurs="unbounded"/>
4212                 </sequence>
4213                 <attribute name="name" type="QName" use="required"/>
4214                 <attribute name="requires" type="sca:listOfQNames"
4215 use="optional"/>
4216                 <attribute name="policySets" type="sca:listOfQNames"
4217                     use="optional"/>
4218                 <anyAttribute namespace="##other" processContents="lax"/>
4219             </extension>
4220         </complexContent>
4221     </complexType>
4222 </schema>
4223

```

4224 B.8 sca-definitions.xsd

```

4225
4226 <?xml version="1.0" encoding="UTF-8"?>
4227 <!-- (c) Copyright SCA Collaboration 2006 -->

```

```
4228 <schema xmlns="http://www.w3.org/2001/XMLSchema"
4229     targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4230     xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4231     elementFormDefault="qualified">
4232
4233     <include schemaLocation="sca-core.xsd"/>
4234
4235     <element name="definitions">
4236         <complexType>
4237             <choice minOccurs="0" maxOccurs="unbounded">
4238                 <element ref="sca:intent"/>
4239                 <element ref="sca:policySet"/>
4240                 <element ref="sca:binding"/>
4241                 <element ref="sca:bindingType"/>
4242                 <element ref="sca:implementationType"/>
4243                 <any namespace="##other" processContents="lax" minOccurs="0"
4244                     maxOccurs="unbounded" />
4245             </choice>
4246         </complexType>
4247     </element>
4248
4249 </schema>
4250
```

4251 **B.9 sca-binding-webservice.xsd**

4252 Is described in [the SCA Web Services Binding specification \[9\]](#)

4253 **B.10 sca-binding-jms.xsd**

4254 Is described in [the SCA JMS Binding specification \[11\]](#)

4255 **B.11 sca-policy.xsd**

4256 Is described in [the SCA Policy Framework specification \[10\]](#)

4257

4258 **B.12 sca-contribution.xsd**

4259

```
4260 <?xml version="1.0" encoding="UTF-8"?>
4261 <!-- (c) Copyright SCA Collaboration 2007 -->
4262 <schema xmlns="http://www.w3.org/2001/XMLSchema"
4263     targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
4264     xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
4265     elementFormDefault="qualified">
4266
4267     <include schemaLocation="sca-core.xsd"/>
4268
4269
```

Comment [mbgl18]: Issue 28

Deleted: 23 September


```

4270     <element name="contribution" type="sca:ContributionType"/>
4271     <complexType name="ContributionType">
4272         <sequence>
4273             <element name="deployable" type="sca:DeployableType"
4274 minOccurs="1" maxOccurs="unbounded"/>
4275             <element name="import" type="sca:ImportType" minOccurs="0"
4276 maxOccurs="unbounded"/>
4277             <element name="export" type="sca:ExportType" minOccurs="0"
4278 maxOccurs="unbounded"/>
4279             <any namespace="##other" processContents="lax" minOccurs="0"
4280 maxOccurs="unbounded"/>
4281         </sequence>
4282         <anyAttribute namespace="##other" processContents="lax"/>
4283     </complexType>
4284
4285
4286
4287     <complexType name="DeployableType">
4288         <sequence>
4289             <any namespace="##other" processContents="lax" minOccurs="0"
4290 maxOccurs="unbounded"/>
4291         </sequence>
4292         <attribute name="composite" type="QName" use="required"/>
4293         <anyAttribute namespace="##other" processContents="lax"/>
4294     </complexType>
4295
4296
4297     <complexType name="ImportType">
4298         <sequence>
4299             <any namespace="##other" processContents="lax" minOccurs="0"
4300 maxOccurs="unbounded"/>
4301         </sequence>
4302         <attribute name="namespace" type="string" use="required"/>
4303         <attribute name="location" type="anyURI" use="required"/>
4304         <anyAttribute namespace="##other" processContents="lax"/>
4305     </complexType>
4306
4307     <complexType name="ExportType">
4308         <sequence>
4309             <any namespace="##other" processContents="lax" minOccurs="0"
4310 maxOccurs="unbounded"/>
4311         </sequence>
4312         <attribute name="namespace" type="string" use="required"/>
4313         <anyAttribute namespace="##other" processContents="lax"/>
4314     </complexType>
4315 </schema>
4316
4317

```

4318 C. SCA Concepts

4319 C.1 Binding

4320 **Bindings** are used by services and references. References use bindings to describe the access
4321 mechanism used to call the service to which they are wired. Services use bindings to describe the
4322 access mechanism(s) that clients should use to call the service.

4323 SCA supports multiple different types of bindings. Examples include **SCA service**, **Web service**,
4324 **stateless session EJB**, **data base stored procedure**, **EIS service**. SCA provides an extensibility
4325 mechanism by which an SCA runtime can add support for additional binding types.

4326

4327 C.2 Component

4328 **SCA components** are configured instances of **SCA implementations**, which provide and consume
4329 services. SCA allows many different implementation technologies such as Java, BPEL, C++. SCA defines
4330 an **extensibility mechanism** that allows you to introduce new implementation types. The current
4331 specification does not mandate the implementation technologies to be supported by an SCA run-time,
4332 vendors may choose to support the ones that are important for them. A single SCA implementation may
4333 be used by multiple Components, each with a different configuration.

4334 The Component has a reference to an implementation of which it is an instance, a set of property values,
4335 and a set of service reference values. Property values define the values of the properties of the
4336 component as defined by the component's implementation. Reference values define the services that
4337 resolve the references of the component as defined by its implementation. These values can either be a
4338 particular service of a particular component, or a reference of the containing composite.

4339 C.3 Service

4340 **SCA services** are used to declare the externally accessible services of an **implementation**. For a
4341 composite, a service is typically provided by a service of a component within the composite, or by a
4342 reference defined by the composite. The latter case allows the republication of a service with a new
4343 address and/or new bindings. The service can be thought of as a point at which messages from external
4344 clients enter a composite or implementation.

4345 A service represents an addressable set of operations of an implementation that are designed to be
4346 exposed for use by other implementations or exposed publicly for use elsewhere (eg public Web services
4347 for use by other organizations). The operations provided by a service are specified by an Interface, as
4348 are the operations required by the service client (if there is one). An implementation may contain
4349 multiple services, when it is possible to address the services of the implementation separately.

4350 A service may be provided **as SCA remote services, as Web services, as stateless session EJB's, as**
4351 **EIS services, and so on**. Services use **bindings** to describe the way in which they are published. SCA
4352 provides an **extensibility mechanism** that makes it possible to introduce new binding types for new
4353 types of services.

4354 C.3.1 Remotable Service

4355 A Remotable Service is a service that is designed to be published remotely in a loosely-coupled SOA
4356 architecture. For example, SCA services of SCA implementations can define implementations of industry-
4357 standard web services. Remotable services use pass-by-value semantics for parameters and returned
4358 results.

4359 A service is remotable if it is defined by a WSDL port type or if it defined by a Java interface marked with
4360 the @Remotable annotation.

4361 C.3.2 Local Service

4362 Local services are services that are designed to be only used “locally” by other implementations that are
4363 deployed concurrently in a tightly-coupled architecture within the same operating system process.

4364 Local services may rely on by-reference calling conventions, or may assume a very fine-grained
4365 interaction style that is incompatible with remote distribution. They may also use technology-specific data-
4366 types.

4367 Currently a service is local only if it defined by a Java interface not marked with the @Remotable
4368 annotation.

4369

4370 C.4 Reference

4371 **SCA references** represent a dependency that an implementation has on a service that is supplied by
4372 some other implementation, where the service to be used is specified through configuration. In other
4373 words, a reference is a service that an implementation may call during the execution of its business
4374 function. References are typed by an interface.

4375 For composites, composite references can be accessed by components within the composite like any
4376 service provided by a component within the composite. Composite references can be used as the targets
4377 of wires from component references when configuring Components.

4378 A composite reference can be used to access a service such as: an SCA service provided by another
4379 SCA composite, a Web service, a stateless session EJB, a data base stored procedure or an EIS service,
4380 and so on. References use **bindings** to describe the access method used to their services. SCA provides
4381 an **extensibility mechanism** that allows the introduction of new binding types to references.

4382

4383 C.5 Implementation

4384 An implementation is concept that is used to describe a piece of software technology such as a Java
4385 class, BPEL process, XSLT transform, or C++ class that is used to implement one or more services in a
4386 service-oriented application. An SCA composite is also an implementation.

4387 Implementations define points of variability including properties that can be set and settable references to
4388 other services. The points of variability are configured by a component that uses the implementation. The
4389 specification refers to the configurable aspects of an implementation as its **componentType**.

4390 C.6 Interface

4391 **Interfaces** define one or more business functions. These business functions are provided by Services
4392 and are used by components through References. Services are defined by the Interface they implement.
4393 SCA currently supports a number of interface type systems, for example:

- 4394 • Java interfaces
- 4395 • WSDL portTypes
- 4396 • C, C++ header files

4397

4398 SCA also provides an extensibility mechanism by which an SCA runtime can add support for additional
4399 interface type systems.

4400 Interfaces may be **bi-directional**. A bi-directional service has service operations which must be provided
4401 by each end of a service communication – this could be the case where a particular service requires a
4402 “callback” interface on the client, which is calls during the process of handing service requests from the
4403 client.

4404

4405 C.7 Composite

4406 An SCA composite is the basic unit of composition within an SCA Domain. An **SCA Composite** is an
4407 assembly of Components, Services, References, and the Wires that interconnect them. Composites can
4408 be used to contribute elements to an **SCA Domain**.

4409 A **composite** has the following characteristics:

- 4410 • It may be used as a component implementation. When used in this way, it defines a boundary for
4411 Component visibility. Components may not be directly referenced from outside of the composite
4412 in which they are declared.
- 4413 • It can be used to define a unit of deployment. Composites are used to contribute business logic
4414 artifacts to an SCA domain.

4415

4416 C.8 Composite inclusion

4417 One composite can be used to provide part of the definition of another composite, through the process of
4418 inclusion. This is intended to make team development of large composites easier. Included composites
4419 are merged together into the using composite at deployment time to form a single logical composite.

4420 Composites are included into other composites through `<include.../>` elements in the using composite.
4421 The SCA Domain uses composites in a similar way, through the deployment of composite files to a
4422 specific location.

4423

4424 C.9 Property

4425 **Properties** allow for the configuration of an implementation with externally set data values. The data
4426 value is provided through a Component, possibly sourced from the property of a containing composite.

4427 Each Property is defined by the implementation. Properties may be defined directly through the
4428 implementation language or through annotations of implementations, where the implementation language
4429 permits, or through a componentType file. A Property can be either a simple data type or a complex data
4430 type. For complex data types, XML schema is the preferred technology for defining the data types.

4431

4432 C.10 Domain

4433 An SCA Domain represents a set of Services providing an area of Business functionality that is controlled
4434 by a single organization. As an example, for the accounts department in a business, the SCA Domain
4435 might cover all finance-related functions, and it might contain a series of composites dealing with specific
4436 areas of accounting, with one for Customer accounts, another dealing with Accounts Payable.

4437 A domain specifies the instantiation, configuration and connection of a set of components, provided via
4438 one or more composite files. The domain, like a composite, also has Services and References. Domains
4439 also contain Wires which connect together the Components, Services and References.

4440

4441 C.11 Wire

4442 **SCA wires** connect **service references** to **services**.

4443 Within a composite, valid wire sources are component references and composite services. Valid wire
4444 targets are component services and composite references.

4445 When using included composites, the sources and targets of the wires don't have to be declared in the
4446 same composite as the composite that contains the wire. The sources and targets can be defined by
4447 other included composites. Targets can also be external to the SCA domain.

4448

4449
4450
4451

D. Conformance Items

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

Conformance ID	Description
ASM10001	An SCA runtime MUST reject a composite file that does not conform to the sca-core.xsd schema.
ASM40002	If present, the @constrainingType attribute of a <componentType/> element MUST reference a <constrainingType/> element in the Domain through its QName.
[Error! Reference source not found.]	The @name attribute of a <service/> child element of a <componentType/> MUST be unique amongst the service elements of that <componentType/>.
[Error! Reference source not found.]	The @name attribute of a <reference/> child element of a <componentType/> MUST be unique amongst the reference elements of that <componentType/>.
ASM40005	The @name attribute of a <property/> child element of a <componentType/> MUST be unique amongst the property elements of that <componentType/>.
ASM40006	If @wiredByImpl is set to "true", then any reference targets configured for this reference MUST be ignored by the runtime.
ASM40007	The value of the property @type attribute MUST be the QName of an XML schema type.
ASM40008	The value of the property @element attribute MUST be the QName of an XSD global element.
ASM40009	The SCA runtime MUST ensure that any implementation default property value is replaced by a value for that property explicitly set by a component using that implementation.
ASM50001	The @name attribute of a <service/> child element of a <componentType/> MUST be unique amongst the service elements of that <componentType/>.
ASM50002	The @name attribute of a service element of a <component/> MUST be unique amongst the service elements of that <component/>
ASM50003	The @name attribute of a service element of a <component/> MUST match the @name attribute of a service element of the componentType of the <implementation/> child element of the component.
ASM50004	If a <service/> element has an interface subelement specified, the interface MUST provide a compatible subset of the interface declared on the componentType of the implementation
ASM50005	If no binding elements are specified for the service, then the bindings specified for the equivalent service in the componentType of the implementation MUST be used, but if the componentType also has no bindings specified, then <binding.sca/> MUST be used as the binding. If binding elements are specified for the service, then those bindings MUST be used and they override any bindings specified for the equivalent service in the componentType of the implementation.
If the callback	If the callback element is present and contains one or more binding child elements,

Formatted: Font: Bold

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Deleted: 23 September

<u>element is present and contains one or more binding child elements, then those bindings MUST be used for the callback.</u> <u>[ASM50006]</u>	<u>then those bindings MUST be used for the callback.</u>
<u>[ASM50007]</u>	<u>The @name attribute of a service element of a <component/> MUST be unique amongst the service elements of that <component/></u>
<u>[ASM50008]</u>	<u>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.</u>
<u>[ASM50009]</u>	<u>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.</u>
<u>[ASM50010]</u>	<u>If @wiredByImpl="true" is set for a reference, then the reference MUST NOT be wired statically within a composite, but left unwired.</u>
<u>[ASM50011]</u>	<u>If an interface is declared for a component reference it MUST provide a compatible superset of the interface declared for the equivalent reference in the componentType of the implementation, i.e. provide the same operations or a superset of the operations defined by the implementation for the reference.</u>
<u>[ASM50012]</u>	<u>If no binding elements are specified for the reference, then the bindings specified for the equivalent reference in the componentType of the implementation MUST be used, but if the componentType also has no bindings specified, then <binding.sca/> MUST be used as the binding. If binding elements are specified for the reference, then those bindings MUST be used and they override any bindings specified for the equivalent reference in the componentType of the implementation.</u>
<u>[ASM50013]</u>	<u>If @wiredByImpl="true", other methods of specifying the target service MUST NOT be used.</u>
<u>[ASM50014]</u>	<u>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.</u>
<u>[ASM50015]</u>	<u>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.</u>
<u>[ASM50016]</u>	<u>It is possible that a particular binding type MAY require that the address of a target service uses more than a simple URI. In such cases, the @uri attribute MUST NOT be used to identify the target service - instead, binding specific attributes and/or child elements must be used.</u>
<u>[ASM50017]</u>	<u>When the reference has a value specified in its @target attribute, one of the child binding elements MUST be used on each wire created by the @target attribute, or the sca binding, if no binding is specified.</u>
<u>[ASM50018]</u>	<u>A reference with multiplicity 0..1 or 0..n MAY have no target service defined.</u>

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Deleted: 23 September

<u>[ASM50019]</u>	<u>A reference with multiplicity 0..1 or 1..1 MUST NOT have more that one target service defined.</u>
<u>[ASM50020]</u>	<u>A reference with multiplicity 1..1 or 1..n MUST have at least one target service defined.</u>
<u>[ASM50021]</u>	<u>A reference with multiplicity 0..n or 1..n MAY have one or more target services defined.</u>
<u>[ASM50022]</u>	<u>Where it is detected that the rules for the number of target services for a reference have been violated, either at deployment or at execution time, an SCA Runtime MUST generate an error no later than when the reference is invoked by the component implementation.</u>
<u>[ASM50023]</u>	<u>Some reference multiplicity errors can be detected at deployment time. In these cases, an error SHOULD be generated by the SCA runtime at deployment time.</u>
<u>[ASM50024]</u>	<u>Other reference multiplicity errors can only be checked at runtime. In these cases, the SCA runtime MUST generate an error no later than when the reference is invoked by the component implementation.</u>
<u>[ASM50025]</u>	<u>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.</u>
<u>[ASM50026]</u>	<u>If a reference has a value specified for one or more target services in its @target attribute, the child binding elements of that reference MUST NOT identify target services using the @uri attribute or using binding specific attributes or elements.</u>

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

4453

E. Acknowledgements

4454
4455

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

4456

Participants:

4457
4458

[Participant Name, Affiliation | Individual Member]
[Participant Name, Affiliation | Individual Member]

4459

F. Non-Normative Text

Deleted: 23 September

4461
4462
4463

G. Revision History

[optional; should not be included in OASIS Standards]

Revision	Date	Editor	Changes Made
1	2007-09-24	Anish Karmarkar	Applied the OASIS template + related changes to the Submission
2	2008-01-04	Michael Beisiegel	<div>composite section</div> <ul style="list-style-type: none">- 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 <div></div> <ul style="list-style-type: none">- moved component section after implementation section- made the ConstrainingType section a top level section- moved interface section to after constraining type section <div>component section</div> <ul style="list-style-type: none">- 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) <div>implementation section</div> <ul style="list-style-type: none">- 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

Deleted: 23 September

			<p>on services, references, and properties should go here</p> <ul style="list-style-type: none"> - 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 <p>"This specification is defined 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."</p>
3	2008-02-15	Anish Karmarkar Michael Beisiegel	<p>Incorporated resolutions from 2008 Jan f2f.</p> <ul style="list-style-type: none"> - issue 9 - issue 19 - issue 21 - issue 4 - issue 1A - issue 27 <ul style="list-style-type: none"> - 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 authoritative
4	2008-04-22	Mike Edwards	<p>Editorial tweaks for CD01 publication:</p> <ul style="list-style-type: none"> - 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	<p>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</p>
6	2008-09-23	Mike Edwards	<p>Editorial fixes in response to Mark Combellack's review contained in email: http://lists.oasis-open.org/archives/sca-assembly/200804/msg00089.html</p>

Page 29: [1] Deleted	Mike Edwards	10/22/2008 10:18:00 AM
If a reference has a value specified for one or more target services in its @target attribute, the child binding elements of that reference MUST NOT identify target services using the @uri attribute or using binding specific attributes or =elements.		
Page 29: [2] Deleted	Mike Edwards	10/22/2008 10:18:00 AM
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.		
Page 29: [3] Deleted	Mike Edwards	10/22/2008 10:20:00 AM
It is possible that a particular binding type MAY require that the address of a target service uses more than a simple URI. In such cases, the @uri attribute MUST NOT be used to identify the target service - instead, binding specific attributes and/or child elements must be used.		
Page 29: [4] Deleted	Mike Edwards	10/22/2008 10:21:00 AM
When the reference has a value specified in its @target attribute, one of the child binding elements MUST be used on each wire created by the @target attribute, or the sca binding, if no binding is specified.		
Page 29: [5] Deleted	Mike Edwards	10/22/2008 10:22:00 AM
A reference with multiplicity 0..1 or 0..n MAY have no target service defined.		
Page 29: [6] Deleted	Mike Edwards	10/22/2008 10:22:00 AM
A reference with multiplicity 0..1 or 1..1 MUST NOT have more than one target service defined.		
Page 29: [7] Deleted	Mike Edwards	10/22/2008 10:23:00 AM
A reference with multiplicity 1..1 or 1..n MUST have at least one target service defined.		
Page 29: [8] Deleted	Mike Edwards	10/22/2008 10:24:00 AM
A reference with multiplicity 0..n or 1..n MAY have one or more target services defined.		