



---

# Service Component Architecture Assembly Model Specification Version 1.1

**Committee Draft 01 Revision 2**

**23rd September, 2008**

**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 .....	24
5.1	Implementation .....	25
5.2	Service .....	26
5.3	Reference .....	27
5.3.1	Specifying the Target Service(s) for a Reference .....	29
5.4	Property .....	31
5.5	Example Component .....	34
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

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

B.7	sca-implementation-composite.xsd.....	111
B.8	sca-definitions.xsd.....	111
B.9	sca-binding-webservice.xsd.....	112
B.10	sca-binding-jms.xsd.....	112
B.11	sca-policy.xsd .....	112
B.12	sca-contribution.xsd .....	112
C.	SCA Concepts .....	114
C.1	Binding .....	114
C.2	Component .....	114
C.3	Service .....	114
C.3.1	Remotable Service.....	114
C.3.2	Local Service .....	115
C.4	Reference .....	115
C.5	Implementation .....	115
C.6	Interface .....	115
C.7	Composite .....	116
C.8	Composite inclusion .....	116
C.9	Property.....	116
C.10	Domain .....	116
C.11	Wire.....	116
D.	Acknowledgements .....	117
E.	Non-Normative Text .....	118
F.	Revision History .....	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_JavaComponentImplementation_V100.pdf)

[http://www.osoa.org/download/attachments/35/SCA\\_JavaAnnotationsAndAPIs\\_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](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](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](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](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](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

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

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

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

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

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

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

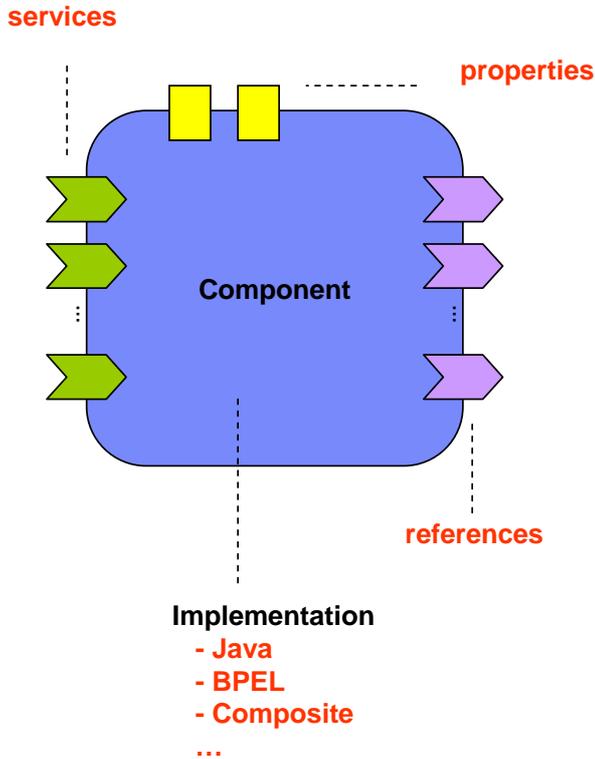
101 Composites are deployed within an **SCA Domain**. An SCA Domain typically represents a set of services  
102 providing an area of business functionality that is controlled by a single organization. As an example, for  
103 the accounts department in a business, the SCA Domain might cover all financial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

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

113 **2.1 Diagram used to Represent SCA Artifacts**

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

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



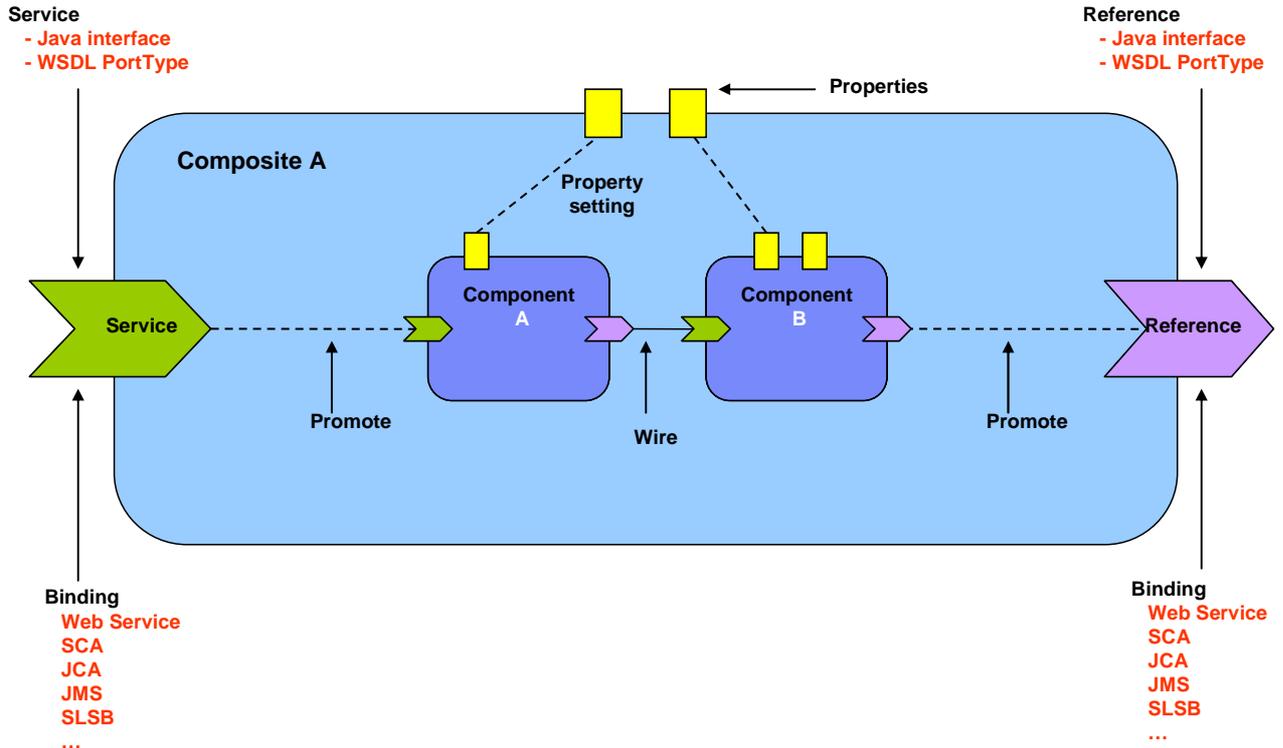
118

119 *Figure 1: SCA Component Diagram*

120

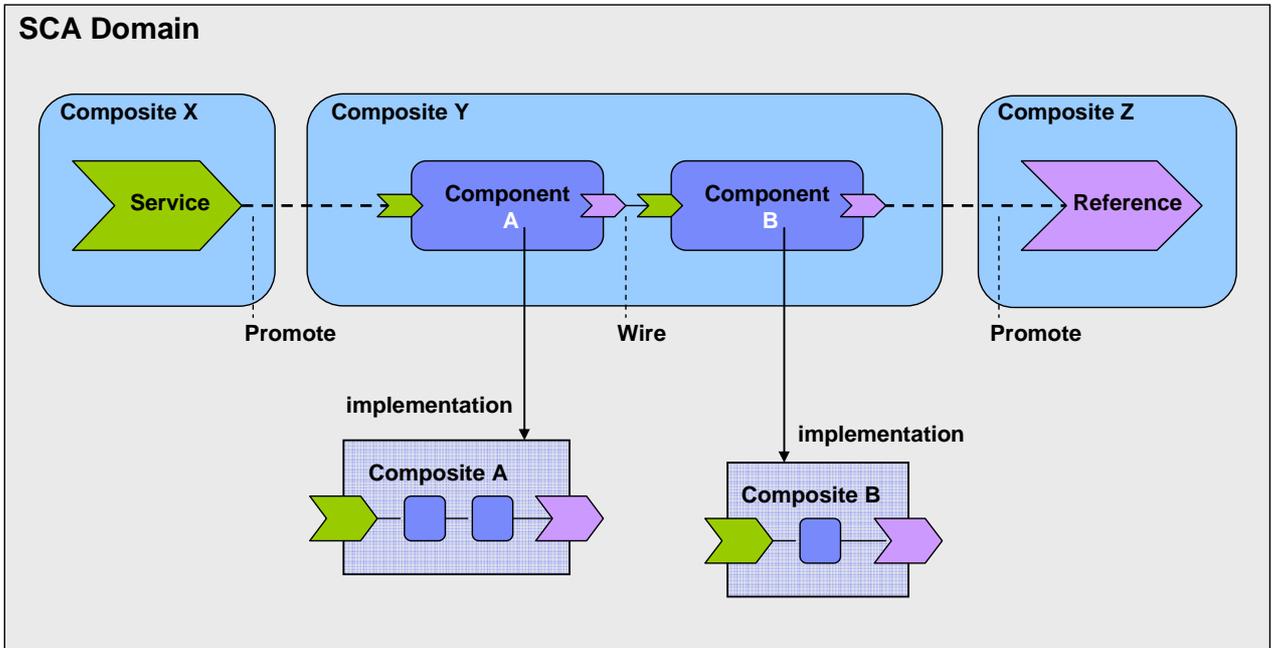
121 The following picture illustrates some of the features of a composite assembled using a set of  
122 components:

123



124  
125 *Figure 2: SCA Composite Diagram*

126  
127 The following picture illustrates an SCA Domain assembled from a series of high-level composites, some  
128 of which are in turn implemented by lower-level composites:



129  
130 *Figure 3: SCA Domain Diagram*

---

131 **3 Quick Tour by Sample**

132 To be completed.

133

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

135

136

137  
138  
139  
140  
141  
142  
143  
144  
145  
146  
147  
148  
149  
150  
151  
152  
153  
154  
155  
156  
157  
158  
159  
160  
161  
162  
163  
164  
165  
166  
167  
168  
169  
170  
171  
172  
173  
174  
175  
176  
177  
178  
179  
180  
181  
182  
183

---

## 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 may have some settable property values.

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

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

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

So, for example:

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

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

### 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.

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

188

189 The following snippet shows the componentType schema.

190

```
191 <?xml version="1.0" encoding="ASCII"?>
192 <!-- Component type schema snippet -->
193 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
194     constrainingType="QName"? >
195
196     <service ... />*
197     <reference ... />*
198     <property ... />*
199     <implementation ... />?
200
201 </componentType>
202
```

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

- 204 • **constrainingType : QName (0..1)** – the name of a constrainingType. When specified,  
205 the set of services, references and properties of the implementation, plus related intents,  
206 is constrained to the set defined by the constrainingType. See [the ConstrainingType](#)  
207 [Section](#) for more details.

208

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

- 210 • **service : Service (0..n)** – see [component type service section](#).
- 211 • **reference : Reference (0..n)** – see [component type reference section](#).
- 212 • **property : Property (0..n)** – see [component type property section](#).
- 213 • **implementation : Implementation (0..1)** – see [component type implementation](#)  
214 [section](#).

215

## 216 4.1.1 Service

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

221

```
222 <?xml version="1.0" encoding="ASCII"?>
223 <!-- Component type service schema snippet -->
224 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ...
225 >
226
227     <service name="xs:NCName"
```

```

228         requires="list of xs:QName"? policySets="list of xs:QName"?>*
229     <interface ... />
230     <binding ... />*
231     <callback>?
232         <binding ... />+
233     </callback>
234 </service>

235
236 <reference ... />*
237 <property ... />*
238 <implementation ... />?
239
240 </componentType>
241

```

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

- 243 • **name : NCName (1..1)** - the name of the service.
- 244 • **requires : QName (0..n)** - a list of policy intents. See the [Policy Framework specification \[10\]](#) for a description of this attribute.
- 245 • **policySets : QName (0..n)** - a list of policy sets. See the [Policy Framework specification \[10\]](#) for a description of this attribute.

248

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

- 250 • **interface : Interface (1..1)** - A service has **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. For details on the interface element see [the Interface section](#).
- 251 • **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](#). 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 \[10\]](#).
- 252 • **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.

264

## 265 4.1.2 Reference

266 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:

271

```

272 <?xml version="1.0" encoding="ASCII"?>
273 <!-- Component type reference schema snippet -->

```

```

274 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ...
275 >
276
277     <service ... />*
278
279     <reference name="xs:NCName"
280         target="list of xs:anyURI"? autowire="xs:boolean"?
281         multiplicity="0..1 or 1..1 or 0..n or 1..n"?
282         wiredByImpl="xs:boolean"?
283         requires="list of xs:QName"? policySets="list of xs:QName"?>*
284     <interface ... />
285     <binding ... />*
286     <callback>?
287         <binding ... />+
288     </callback>
289 </reference>
290
291 <property ... />*
292 <implementation ... />?
293
294 </componentType>
295

```

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

- 297 • **name : NCName (1..1)** - the name of the reference.
- 298 • **multiplicity : 0..1|1..1|0..n|1..n (0..1)** - defines the number of wires that can connect  
299 the reference to target services. The multiplicity can have the following values
  - 300 ○ 0..1 – zero or one wire can have the reference as a source
  - 301 ○ 1..1 – one wire can have the reference as a source
  - 302 ○ 0..n - zero or more wires can have the reference as a source
  - 303 ○ 1..n – one or more wires can have the reference as a source
- 304 • **target : anyURI (0..n)** - a list of one or more of target service URI's, depending on  
305 multiplicity setting. Each value wires the reference to a component service that resolves  
306 the reference. For more details on wiring see [the section on Wires](#).
- 307 • **autowire : boolean (0..1)** - whether the reference should be autowired, as described in  
308 [the Autowire section](#). Default is false.
- 309 • **wiredByImpl : boolean (0..1)** - a boolean value, "false" by default, which indicates that  
310 the implementation wires this reference dynamically. If set to "true" it indicates that the  
311 target of the reference is set at runtime by the implementation code (eg by the code  
312 obtaining an endpoint reference by some means and setting this as the target of the  
313 reference through the use of programming interfaces defined by the relevant Client and  
314 Implementation specification). If "true" is set, then the reference should not be wired  
315 statically within a composite, but left unwired.
- 316 • **requires : QName (0..n)** - a list of policy intents. See the [Policy Framework specification](#)  
317 [\[10\]](#) for a description of this attribute.
- 318 • **policySets : QName (0..n)** - a list of policy sets. See the [Policy Framework specification](#)  
319 [\[10\]](#) for a description of this attribute.

320  
321  
322  
323  
324  
325  
326  
327  
328  
329  
330  
331  
332  
333  
334  
335  
336  
337  
338  
339  
340  
341  
342  
343

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](#). The binding, combined with any PolicySets in effect for the binding, must satisfy the set of policy intents for the reference, as described in [the Policy Framework specification \[10\]](#).  
  
Note that a binding element may specify an endpoint which is the target of that binding. A reference must not mix the use of endpoints specified via binding elements with target endpoints specified via the target attribute. If the target attribute is set, then binding elements can only list one or more binding types that can be used for the wires identified by the target attribute. All the binding types identified are available for use on each wire in this case. If endpoints are specified in the binding elements, each endpoint must use the binding type of the binding element in which it is defined. In addition, each binding element needs to specify an endpoint in this case.
- **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.

### 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 ... />?
```

367 </componentType>  
368

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

- 370 ▪ **name : NCName (1..1)** - the name of the property.
- 371 ▪ one of **(1..1)**:
  - 372 ○ **type : QName** - the type of the property defined as the qualified name of an XML  
373 schema type.
  - 374 ○ **element : QName** - the type of the property defined as the qualified name of an  
375 XML schema global element – the type is the type of the global element.
- 376 ▪ **many : boolean (0..1)** - (optional) whether the property is single-valued (false) or multi-  
377 valued (true). In the case of a multi-valued property, it is presented to the implementation  
378 as a collection of property values.
- 379 ▪ **mustSupply : boolean (0..1)** - whether the property value must be supplied by the  
380 component that uses the implementation – when mustSupply="true" the component must  
381 supply a value since the implementation has no default value for the property. A default-  
382 property-value should only be supplied when mustSupply="false" (the default setting for  
383 the mustSupply attribute), since the implication of a default value is that it is used only  
384 when a value is not supplied by the using component.
- 385 ▪ **file : anyURI (0..1)** - a dereferencable URI to a file containing a value for the property.

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

391

392 The componentType property element MAY contain an SCA default value for the property declared  
393 by the implementation. However, the implementation MAY have a property which has an  
394 implementation defined default value, where the default value is not represented in the  
395 componentType. An example of such a default value is where the default value is computed at  
396 runtime by some code contained in the implementation. If a using component needs to control the  
397 value of a property used by an implementation, the component MUST set the value explicitly and  
398 the SCA runtime MUST ensure that any implementation default value is replaced.

399

400

#### 401 4.1.4 Implementation

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

406

```
407 <?xml version="1.0" encoding="ASCII"?>  
408 <!-- Component type implementation schema snippet -->  
409 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ...  
410 >  
411  
412     <service ... />*  
413     <reference ... >*
```

```

414     <property ... />*
415
416     <implementation requires="list of xs:QName"?
417         policySets="list of xs:QName"?/>?
418
419 </componentType>
420

```

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

- 422 • **requires : QName (0..n)** - a list of policy intents. See the [Policy Framework specification \[10\]](#) for a description of this attribute.
- 423
- 424 • **policySets : QName (0..n)** - a list of policy sets. See the [Policy Framework specification \[10\]](#) for a description of this attribute.
- 425

426  
427

## 428 4.2 Example ComponentType

429

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

433

```

434 <?xml version="1.0" encoding="ASCII"?>
435 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
436
437     <service name="MyValueService">
438         <interface.java interface="services.myvalue.MyValueService"/>
439     </service>
440
441     <reference name="customerService">
442         <interface.java interface="services.customer.CustomerService"/>
443     </reference>
444     <reference name="stockQuoteService">
445         <interface.java
446 interface="services.stockquote.StockQuoteService"/>
447     </reference>
448
449     <property name="currency" type="xsd:string">USD</property>
450
451 </componentType>
452

```

## 453 4.3 Example Implementation

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

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

```
458  
459 package services.account;  
460  
461 @Remotable  
462 public interface AccountService {  
463  
464     AccountReport getAccountReport(String customerID);  
465 }  
466
```

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

```
470  
471 package services.account;  
472  
473 import java.util.List;  
474  
475 import commonj.sdo.DataFactory;  
476  
477 import org.osoa.sca.annotations.Property;  
478 import org.osoa.sca.annotations.Reference;  
479  
480 import services.accountdata.AccountDataService;  
481 import services.accountdata.CheckingAccount;  
482 import services.accountdata.SavingsAccount;  
483 import services.accountdata.StockAccount;  
484 import services.stockquote.StockQuoteService;  
485  
486 public class AccountServiceImpl implements AccountService {  
487  
488     @Property  
489     private String currency = "USD";  
490  
491     @Reference  
492     private AccountDataService accountDataService;  
493     @Reference  
494     private StockQuoteService stockQuoteService;  
495  
496     public AccountReport getAccountReport(String customerID) {  
497  
498         DataFactory dataFactory = DataFactory.INSTANCE;  
499         AccountReport accountReport = (AccountReport)dataFactory.create(AccountReport.class);  
500         List accountSummaries = accountReport.getAccountSummaries();  
501  
502         CheckingAccount checkingAccount = accountDataService.getCheckingAccount(customerID);  
503         AccountSummary checkingAccountSummary =  
504 (AccountSummary)dataFactory.create(AccountSummary.class);  
505         checkingAccountSummary.setAccountNumber(checkingAccount.getAccountNumber());  
506     }  
507 }  
508
```

```

506     checkingAccountSummary.setAccountType("checking");
507     checkingAccountSummary.setBalance(fromUSDollarToCurrency(checkingAccount.getBalance()));
508     accountSummaries.add(checkingAccountSummary);
509
510     SavingsAccount savingsAccount = accountDataService.getSavingsAccount(customerID);
511     AccountSummary savingsAccountSummary =
512 (AccountSummary)dataFactory.create(AccountSummary.class);
513     savingsAccountSummary.setAccountNumber(savingsAccount.getAccountNumber());
514     savingsAccountSummary.setAccountType("savings");
515     savingsAccountSummary.setBalance(fromUSDollarToCurrency(savingsAccount.getBalance()));
516     accountSummaries.add(savingsAccountSummary);
517
518     StockAccount stockAccount = accountDataService.getStockAccount(customerID);
519     AccountSummary stockAccountSummary =
520 (AccountSummary)dataFactory.create(AccountSummary.class);
521     stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());
522     stockAccountSummary.setAccountType("stock");
523     float balance=
524 (stockQuoteService.getQuote(stockAccount.getSymbol()))*stockAccount.getQuantity();
525     stockAccountSummary.setBalance(fromUSDollarToCurrency(balance));
526     accountSummaries.add(stockAccountSummary);
527
528     return accountReport;
529 }
530
531 private float fromUSDollarToCurrency(float value){
532
533     if (currency.equals("USD")) return value; else
534     if (currency.equals("EURO")) return value * 0.8f; else
535     return 0.0f;
536 }
537 }

```

538

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

541

```

542 <?xml version="1.0" encoding="ASCII"?>
543 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
544     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
545
546     <service name="AccountService">
547         <interface.java interface="services.account.AccountService"/>
548     </service>
549     <reference name="accountDataService">
550         <interface.java
551 interface="services.accountdata.AccountDataService"/>
552     </reference>
553     <reference name="stockQuoteService">

```

```
554         <interface.java
555 interface="services.stockquote.StockQuoteService"/>
556     </reference>
557
558     <property name="currency" type="xsd:string">USD</property>
559
560 </componentType>
```

561

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

---

## 565 5 Component

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

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

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

```
575  
576 <?xml version="1.0" encoding="UTF-8"?>  
577 <!-- Component schema snippet -->  
578 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >  
579   ...  
580   <component name="xs:NCName" autowire="xs:boolean"?  
581     requires="list of xs:QName"? policySets="list of xs:QName"?  
582     constrainingType="xs:QName"?>*  
583     <implementation ... />?  
584     <service ... />*  
585     <reference ... />*  
586     <property ... />*  
587   </component>  
588   ...  
589 </composite>  
590
```

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

- 592 • **name : NCName (1..1)** – the name of the component. The name must be unique across  
593 all the components in the composite.
- 594 • **autowire : boolean (0..1)** – whether contained component references should be  
595 autowired, as described in [the Autowire section](#). Default is false.
- 596 • **requires : QName (0..n)** – a list of policy intents. See the [Policy Framework specification](#)  
597 [\[10\]](#) for a description of this attribute.
- 598 • **policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification](#)  
599 [\[10\]](#) for a description of this attribute.
- 600 • **constrainingType : QName (0..1)** – the name of a constrainingType. When specified,  
601 the set of services, references and properties of the component, plus related intents, is  
602 constrained to the set defined by the constrainingType. See [the ConstrainingType Section](#)  
603 for more details.

604

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

- 606 • **implementation : ComponentImplementation (0..1)** – see component  
607 implementation section.

- 608 • **service : ComponentService (0..n)** – see component service section.
- 609 • **reference : ComponentReference (0..n)** – see component reference section.
- 610 • **property : ComponentProperty (0..n)** – see component property section.
- 611

## 612 5.1 Implementation

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

```
618  
619 <?xml version="1.0" encoding="UTF-8"?>  
620 <!-- Component Implementation schema snippet -->  
621 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >  
622   ...  
623   <component ... >*<br>  
624     <implementation ... />?<br>  
625     <service ... />*<br>  
626     <reference ... />*<br>  
627     <property ... />*<br>  
628   </component><br>  
629   ...<br>  
630 </composite><br>  
631
```

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

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

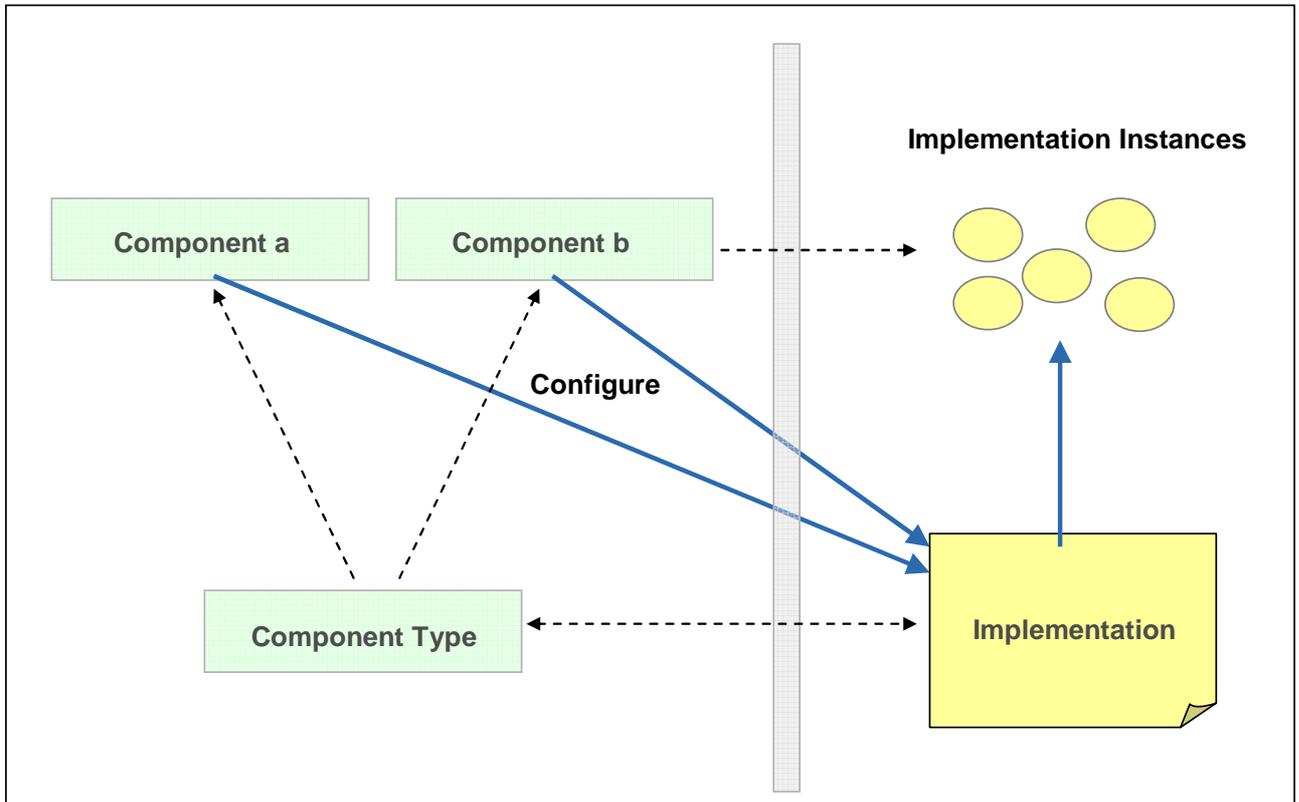
640 The following snippets show implementation elements for the Java and BPEL implementation types  
641 and for the use of a composite as an implementation:

```
642  
643 <implementation.java class="services.myvalue.MyValueServiceImpl" /><br>  
644  
645 <implementation.bpel process="ans:MoneyTransferProcess" /><br>  
646  
647 <implementation.composite name="bns:MyValueComposite" /><br>  
648  
649
```

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

651

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



657  
 658 *Figure 4: Relationship of Component and Implementation*  
 659

## 660 5.2 Service

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

```
665
666 <?xml version="1.0" encoding="UTF-8"?>
667 <!-- Component Service schema snippet -->
668 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
669   ...
670   <component ... >*
671     <implementation ... />?
672     <service name="xs:NCName" requires="list of xs:QName"?
673       policySets="list of xs:QName"?>*
674     <interface ... />?
675     <binding ... />*
```

```

676         <callback?
677             <binding ... />+
678         </callback>
679     </service>
680     <reference ... />*
681     <property ... />*
682 </component>
683 ...
684 </composite>
685

```

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

- 687 • **name : NCName (1..1)** - the name of the service. Has to match a name of a service  
688 defined by the implementation.
- 689 • **requires : QName (0..n)** – a list of policy intents. See the [Policy Framework specification \[10\]](#) for a description of this attribute.  
690 Note: The effective set of policy intents for the service consists of any intents explicitly  
691 stated in this requires attribute, combined with any intents specified for the service by the  
692 implementation.
- 694 • **policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification \[10\]](#) for a description of this attribute.  
695

696

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

- 698 • **interface : Interface (0..1)** - A service has **zero or one interface**, which describes the  
699 operations provided by the service. The interface is described by an **interface element**  
700 which is a child element of the service element. If no interface is specified, then the  
701 interface specified for the service by the implementation is in effect. If an interface is  
702 specified it must provide a compatible subset of the interface provided by the  
703 implementation, i.e. provide a subset of the operations defined by the implementation for  
704 the service. For details on the interface element see [the Interface section](#).
- 705 • **binding : Binding (0..n)** - A service element has **zero or more binding elements** as  
706 children. If no bindings are specified, then the bindings specified for the service by the  
707 implementation are in effect. If bindings are specified, then those bindings override the  
708 bindings specified by the implementation. Details of the binding element are described in  
709 [the Bindings section](#). The binding, combined with any PolicySets in effect for the binding,  
710 must satisfy the set of policy intents for the service, as described in [the Policy Framework specification \[10\]](#).  
711
- 712 • **callback (0..1) / binding : Binding (1..n)** - A service element has an optional **callback**  
713 element used if the interface has a callback defined, which has one or more **binding**  
714 elements as children. The **callback** and its binding child elements are specified if there is  
715 a need to have binding details used to handle callbacks. If the callback element is not  
716 present, the behaviour is runtime implementation dependent.

717

## 718 5.3 Reference

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

723

```

724 <?xml version="1.0" encoding="UTF-8"?>
725 <!-- Component Reference schema snippet -->
726 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
727   ...
728   <component ... >*
729     <implementation ... />?
730     <service ... />*
731     <reference name="xs:NCName"
732       target="list of xs:anyURI"? autowire="xs:boolean"?
733       multiplicity="0..1 or 1..1 or 0..n or 1..n"?
734       wiredByImpl="xs:boolean"? requires="list of xs:QName"?
735       policySets="list of xs:QName"?>*
736     <interface ... />?
737     <binding uri="xs:anyURI"? requires="list of xs:QName"?
738       policySets="list of xs:QName"?/>*
739     <callback?
740       <binding ... />+
741     </callback>
742   </reference>
743   <property ... />*
744 </component>
745   ...
746 </composite>
747

```

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

- 749 • **name : NCName (1..1)** – the name of the reference. Has to match a name of a reference  
750 defined by the implementation.
- 751 • **autowire : boolean (0..1)** – whether the reference should be autowired, as described in  
752 [the Autowire section](#). Default is false.
- 753 • **requires : QName (0..n)** – a list of policy intents. See the [Policy Framework specification](#)  
754 [\[10\]](#) for a description of this attribute.  
755 Note: The effective set of policy intents for the reference consists of any intents explicitly  
756 stated in this requires attribute, combined with any intents specified for the reference by  
757 the implementation.
- 758 • **policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification](#)  
759 [\[10\]](#) for a description of this attribute.
- 760 • **multiplicity : 0..1|1..1|0..n|1..n (0..1)** - defines the number of wires that can connect  
761 the reference to target services. Overrides the multiplicity specified for this reference on  
762 the implementation. The value can only be equal or further restrict, i.e. 0..n to 0..1 or 1..n  
763 to 1..1. The multiplicity can have the following values
  - 764 ○ 0..1 – zero or one wire can have the reference as a source
  - 765 ○ 1..1 – one wire can have the reference as a source
  - 766 ○ 0..n - zero or more wires can have the reference as a source
  - 767 ○ 1..n – one or more wires can have the reference as a source

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

779

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

- 781 • **interface : Interface (0..1)** - A reference has **zero or one interface**, which describes  
782 the operations required by the reference. The interface is described by an **interface**  
783 **element** which is a child element of the reference element. If no interface is specified,  
784 then the interface specified for the reference by the implementation is in effect. If an  
785 interface is specified it must provide a compatible superset of the interface provided by the  
786 implementation, i.e. provide a superset of the operations defined by the implementation  
787 for the reference. For details on the interface element see [the Interface section](#).
- 788 • **binding : Binding (0..n)** - A reference element has **zero or more binding elements** as  
789 children. If no bindings are specified, then the bindings specified for the reference by the  
790 implementation are in effect. If any bindings are specified, then those bindings override  
791 any and all the bindings specified by the implementation. Details of the binding element  
792 are described in the [Bindings section](#). The binding, combined with any PolicySets in effect  
793 for the binding, must satisfy the set of policy intents for the reference, as described in [the](#)  
794 [Policy Framework specification \[10\]](#).  
795 A reference identifies zero or more target services that satisfy the reference. This can be  
796 done in a number of ways, which are fully described in section "5.3.1 Specifying the  
797 Target Service(s) for a Reference"
- 798 • **callback (0..1) / binding : Binding (1..n)** - A **reference** element has an optional  
799 **callback** element used if the interface has a callback defined, which has one or more  
800 **binding** elements as children. The **callback** and its binding child elements are specified if  
801 there is a need to have binding details used to handle callbacks. If the callback element is  
802 not present, the behaviour is runtime implementation dependent.

### 803 5.3.1 Specifying the Target Service(s) for a Reference

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

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

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

- 818 • If @wiredByImpl="true", other methods of specifying the target service MUST NOT be  
819 used.
- 820 • If @autowire="true", the autowire procedure MUST only be used if no target is identified  
821 by any of the other ways listed above. It is not an error if @autowire="true" and a target  
822 is also defined through some other means, however in this case the autowire procedure  
823 MUST NOT be used.
- 824 • If a reference has a value specified for one or more target services in its @target attribute,  
825 the child binding elements of that reference MUST NOT identify target services using the  
826 @uri attribute or using binding specific attributes or =elements.
- 827 • If a binding element has a value specified for a target service using its @uri attribute, the  
828 binding element MUST NOT identify target services using binding specific attributes or  
829 elements.
- 830 • It is possible that a particular binding type MAY require that the address of a target service  
831 uses more than a simple URI. In such cases, the @uri attribute MUST NOT be used to  
832 identify the target service - instead, binding specific attributes and/or child elements must  
833 be used.
- 834 • When the reference has a value specified in its @target attribute, one of the child binding  
835 elements MUST be used on each wire created by the @target attribute, or the sca binding,  
836 if no binding is specified.

### 837 **5.3.1.1 Multiplicity and the Valid Number of Target Services for a Reference**

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

- 839 • A reference with multiplicity 0..1 or 0..n MAY have no target service defined.
- 840 • A reference with multiplicity 0..1 or 1..1 MUST NOT have more than one target service  
841 defined.
- 842 • A reference with multiplicity 1..1 or 1..n MUST have at least one target service defined.
- 843 • A reference with multiplicity 0..n or 1..n MAY have one or more target services defined.

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

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

853 Other errors can only be checked at runtime. Examples include cases of components deployed to  
854 the SCA Domain. At the Domain level, the target of a wire, or even the wire itself, may form part  
855 of a separate deployed contribution and as a result these may be deployed after the original  
856 component is deployed. In these cases, the SCA runtime MUST generate an error no later than  
857 when the reference is invoked by the component implementation.

858 For the cases where it is valid for the reference to have no target service specified, the component  
859 implementation language specification MUST define the programming model for interacting with  
860 an untargetted reference.

861 Where a component reference is promoted by a composite reference, the promotion MUST be  
862 treated from a multiplicity perspective as providing 0 or more target services for the component  
863 reference, depending upon the further configuration of the composite reference. These target  
864 services are in addition to any target services identified on the component reference itself, subject  
865 to the rules relating to multiplicity described in this section.

## 866 5.4 Property

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

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

- 874 • As a value, supplied in the **value** attribute of the property element.  
875 This can be used only when the type of the property is specified using a XML Schema  
876 simple type and a single value is specified.

877 For example,

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

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

883 For example,

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

```
886 <property name="pi">  
887     <value>3.14159265</value>  
888 </property>
```

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

```
891 <property name="currency">  
892     <value>EURO</value>  
893     <value>USDollar</value>  
894 </property>
```

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

```
897 <property name="complexFoo">  
898     <value attr="bar">  
899         <foo:a>TheValue</foo:a>  
900         <foo:b>InterestingURI</foo:b>  
901     </value>  
902 </property>
```

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

```
905 <property name="complexBar">  
906     <value anotherAttr="foo">  
907         <bar:a>AValue</bar:a>  
908         <bar:b>InterestingURI</bar:b>  
909     </value>
```

```

910         <value attr="zing">
911             <bar:a>BValue</bar:a>
912             <bar:b>BoringURI</bar:b>
913         </value>
914     </property>

```

- As a value, supplied as the content of the property element. This can be used only when the type of the property is specified using a XML Schema global element declaration.

918 For example,

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

```

921     <property name="foo">
922         <foo:SomeGED ...>...</foo:SomeGED>
923     </property>

```

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

```

926     <property name="bar">
927         <bar:SomeOtherGED ...>...</bar:SomeOtherGED>
928         <bar:SomeOtherGED ...>...</bar:SomeOtherGED>
929     </property>

```

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

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

937 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.

943

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

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

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

951 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

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

957  
958  
959  
960  
961  
962  
963  
964  
965  
966  
967  
968  
969  
970  
971  
972  
973  
974  
975  
976  
977  
978  
979  
980  
981  
982  
983  
984  
985  
986  
987  
988  
989  
990  
991  
992  
993  
994  
995  
996  
997  
998  
999  
1000  
1001

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**:

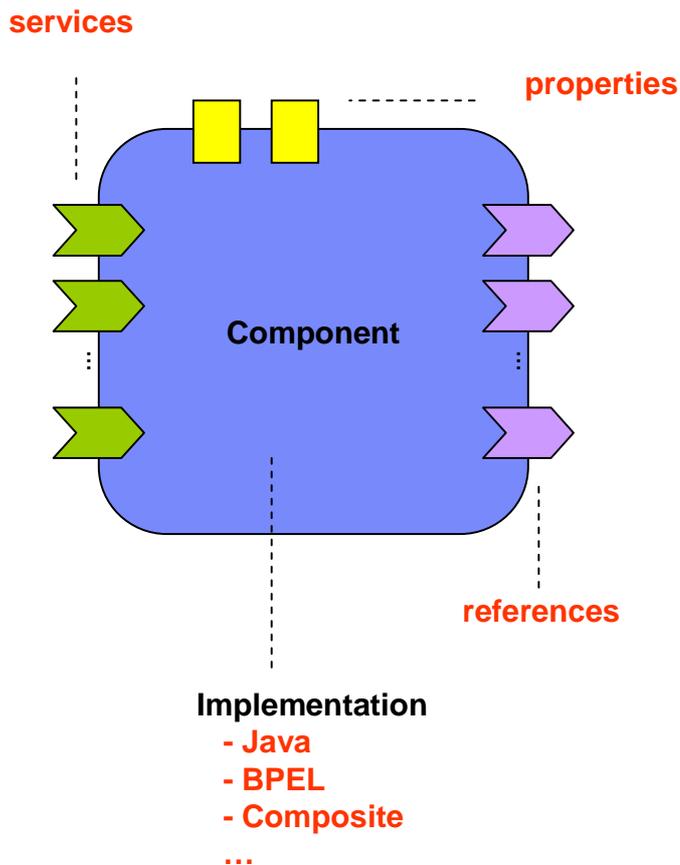
- **name : NCName (1..1)** – the name of the property. Has to match a name of a property defined by the implementation
- zero or one of (**0..1**):
  - **type : QName** – the type of the property defined as the qualified name of an XML schema type
  - **element : QName** – the type of the property defined as the qualified name of an XML schema global element – the type is the type of the global element
- **source : string (0..1)** – an XPath expression pointing to a property of the containing composite from which the value of this component property is obtained.
- **file : anyURI (0..1)** – a dereferencable URI to a file containing a value for the property
- **many : boolean (0..1)** – (optional) whether the property is single-valued (false) or multi-valued (true). Overrides the many specified for this property on the implementation. The value can only be equal or further restrict, i.e. if the implementation specifies many true, then the component can say false. In the case of a multi-valued property, it is presented to the implementation as a Collection of property values.
- **mustSupply : boolean (0..1)** - whether the property value must be supplied by the component – when mustSupply="true" the component must supply a value since the implementation has no default value for the property.
- **value : string (0..1)** - the value of the property if the property is defined using a simple type. This property cannot be used if multiple values are specified (for multivalued properties).

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

1002 **value :any (0..n)** - A property has **zero or more**, value elements that specify the value(s) of a  
1003 property that is defined using a XML Schema type. If the property is single-valued, this element  
1004 MUST NOT occur more than once. This element MUST NOT be used when the @value attribute is  
1005 used to specify the property value.

## 5.5 Example Component

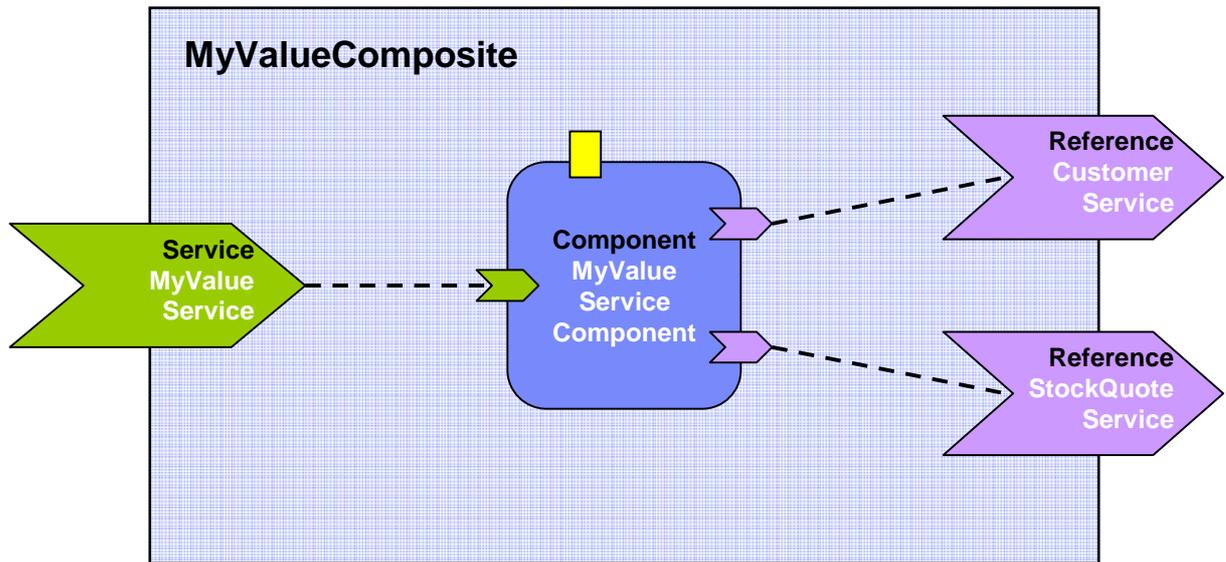
1006  
1007  
1008 The following figure shows the **component symbol** that is used to represent a component in an  
1009 assembly diagram.



1010  
1011 *Figure 5: Component symbol*

1012 The following figure shows the assembly diagram for the MyValueComposite containing the  
1013 MyValueServiceComponent.

1014



1015  
1016  
1017  
1018

Figure 6: Assembly diagram for MyValueComposite

1019 The following snippet shows the MyValueComposite.composite file for the MyValueComposite  
1020 containing the component element for the MyValueServiceComponent. A value is set for the  
1021 property named currency, and the customerService and stockQuoteService references are  
1022 promoted:

1023  
1024  
1025  
1026  
1027  
1028  
1029  
1030  
1031  
1032  
1033  
1034  
1035  
1036  
1037  
1038  
1039  
1040  
1041  
1042  
1043  
1044

```
<?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"/>

```

1045  
1046  
1047  
1048  
1049  
1050  
1051  
1052  
1053  
1054  
1055  
1056  
1057  
1058  
1059  
1060  
1061  
1062  
1063  
1064  
1065  
1066  
1067  
1068  
1069  
1070  
1071  
1072  
1073  
1074  
1075  
1076  
1077  
1078  
1079  
1080  
1081  
1082  
1083  
1084  
1085  
1086

```
</composite>
```

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

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

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

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

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

    ...

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

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

</composite>
```

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

---

## 1087 6 Composite

1088 An SCA composite is used to assemble SCA elements in logical groupings. It is the basic unit of  
1089 composition within an SCA Domain. An **SCA composite** contains a set of components, services,  
1090 references and the wires that interconnect them, plus a set of properties which can be used to  
1091 configure components.

1092 Composites may form **component implementations** in higher-level composites – in other words  
1093 the higher-level composites can have components that are implemented by composites. For more  
1094 detail on the use of composites as component implementations see the section [Using Composites](#)  
1095 [as Component Implementations](#).

1096 The content of a composite may be used within another composite through **inclusion**. When a  
1097 composite is included by another composite, all of its contents are made available for use within  
1098 the including composite – the contents are fully visible and can be referenced by other elements  
1099 within the including composite. For more detail on the inclusion of one composite into another see  
1100 the section [Using Composites through Inclusion](#).

1101 A composite can be used as a unit of deployment. When used in this way, composites contribute  
1102 elements to an SCA domain. A composite can be deployed to the SCA domain either by inclusion,  
1103 or a composite can be deployed to the domain as an implementation. For more detail on the  
1104 deployment of composites, see the section dealing with the [SCA Domain](#).

1105

1106 A composite is defined in an **xxx.composite** file. A composite is represented by a **composite**  
1107 element. The following snippet shows the schema for the composite element.

1108

```
1109 <?xml version="1.0" encoding="ASCII"?>  
1110 <!-- Composite schema snippet -->  
1111 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
1112     targetNamespace="xs:anyURI"  
1113     name="xs:NCName" local="xs:boolean"?  
1114     autowire="xs:boolean"? constrainingType="QName"?  
1115     requires="list of xs:QName"? policySets="list of xs:QName"?>  
1116  
1117     <include ... /*>  
1118  
1119     <service ... /*>  
1120     <reference ... /*>  
1121     <property ... /*>  
1122  
1123     <component ... /*>  
1124  
1125     <wire ... /*>  
1126  
1127 </composite>
```

1128

1129

1130

1131 The **composite** element has the following **attributes**:

- 1132 • **name : NCName (1..1)** – the name of the composite. The form of a composite name is  
1133 an XML QName, in the namespace identified by the targetNamespace attribute.
- 1134 • **targetNamespace : anyURI (0..1)** – an identifier for a target namespace into which the  
1135 composite is declared
- 1136 • **local : boolean (0..1)** – whether all the components within the composite must all run in  
1137 the same operating system process. local="true" means that all the components must run  
1138 in the same process. local="false", which is the default, means that different components  
1139 within the composite may run in different operating system processes and they may even  
1140 run on different nodes on a network.
- 1141 • **autowire : boolean (0..1)** – whether contained component references should be  
1142 autowired, as described in [the Autowire section](#). Default is false.
- 1143 • **constrainingType : QName (0..1)** – the name of a constrainingType. When specified,  
1144 the set of services, references and properties of the composite, plus related intents, is  
1145 constrained to the set defined by the constrainingType. See [the ConstrainingType Section](#)  
1146 for more details.
- 1147 • **requires : QName (0..n)** – a list of policy intents. See the [Policy Framework](#)  
1148 [specification \[10\]](#) for a description of this attribute.
- 1149 • **policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification](#)  
1150 [\[10\]](#) for a description of this attribute.

1151

1152 The **composite** element has the following **child elements**:

- 1153 • **service : CompositeService (0..n)** – see composite service section.
- 1154 • **reference : CompositeReference (0..n)** – see composite reference section.
- 1155 • **property : CompositeProperty (0..n)** – see composite property section.
- 1156 • **component : Component (0..n)** – see component section.
- 1157 • **wire : Wire (0..n)** – see composite wire section.
- 1158 • **include : Include (0..n)** – see composite include section

1159

1160 Components contain configured implementations which hold the business logic of the composite.  
1161 The components offer services and require references to other services. Composite services  
1162 define the public services provided by the composite, which can be accessed from outside the  
1163 composite. Composite references represent dependencies which the composite has on services  
1164 provided elsewhere, outside the composite. Wires describe the connections between component  
1165 services and component references within the composite. Included composites contribute the  
1166 elements they contain to the using composite.

1167 Composite services involve the **promotion** of one service of one of the components within the  
1168 composite, which means that the composite service is actually provided by one of the components  
1169 within the composite. Composite references involve the **promotion** of one or more references of  
1170 one or more components. Multiple component references can be promoted to the same composite  
1171 reference, as long as all the component references are compatible with one another. Where  
1172 multiple component references are promoted to the same composite reference, then they all share  
1173 the same configuration, including the same target service(s).

1174 Composite services and composite references can use the configuration of their promoted services  
1175 and references respectively (such as Bindings and Policy Sets). Alternatively composite services  
1176 and composite references can override some or all of the configuration of the promoted services  
1177 and references, through the configuration of bindings and other aspects of the composite service  
1178 or reference.

1179 Component services and component references can be promoted to composite services and  
1180 references and also be wired internally within the composite at the same time. For a reference,  
1181 this only makes sense if the reference supports a multiplicity greater than 1.

1182

## 1183 6.1 Service

1184 The **services of a composite** are defined by promoting services defined by components  
1185 contained in the composite. A component service is promoted by means of a composite **service**  
1186 **element**.

1187 A composite service is represented by a **service element** which is a child of the composite  
1188 element. There can be **zero or more** service elements in a composite. The following snippet  
1189 shows the composite schema with the schema for a service child element:

1190

```
1191 <?xml version="1.0" encoding="ASCII"?>
1192 <!-- Composite Service schema snippet -->
1193 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
1194   ...
1195   <service name="xs:NCName" promote="xs:anyURI"
1196     requires="list of xs:QName"? policySets="list of xs:QName"?>*
1197     <interface ... />?
1198     <binding ... />*
1199     <callback?
1200       <binding ... />+
1201     </callback>
1202   </service>
1203   ...
1204 </composite>
```

1205

1206 The **composite service** element has the following **attributes**:

- 1207 • **name : NCName (1..1)** – the name of the service, the name MUST be unique across all  
1208 the composite services in the composite. The name of the composite service can be  
1209 different from the name of the promoted component service.
- 1210 • **promote : anyURI (1..1)** – identifies the promoted service, the value is of the form  
1211 <component-name>/<service-name>. The service name is optional if the target  
1212 component only has one service. The same component service can be promoted by more  
1213 than one composite service.
- 1214 • **requires : QName (0..n)** – a list of required policy intents. See the [Policy Framework](#)  
1215 [specification \[10\]](#) for a description of this attribute. Specified **required intents** add to or  
1216 further qualify the required intents defined by the promoted component service.
- 1217 • **policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification](#)  
1218 [\[10\]](#) for a description of this attribute.

1219

1220 The **composite service** element has the following **child elements**, whatever is not specified is  
1221 defaulted from the promoted component service.

- 1222 • **interface : Interface (0..1)** - If an **interface** is specified it must be the same or a  
1223 compatible subset of the interface provided by the promoted component service, i.e.  
1224 provide a subset of the operations defined by the component service. The interface is

1225 described by **zero or one interface element** which is a child element of the service  
1226 element. For details on the interface element see [the Interface section](#).

1227 • **binding : Binding (0..n)** - If bindings are specified they **override** the bindings defined  
1228 for the promoted component service from the composite service perspective. The bindings  
1229 defined on the component service are still in effect for local wires within the composite  
1230 that target the component service. A service element has zero or more **binding elements**  
1231 as children. Details of the binding element are described in the [Bindings section](#). For more  
1232 details on wiring see [the Wiring section](#).

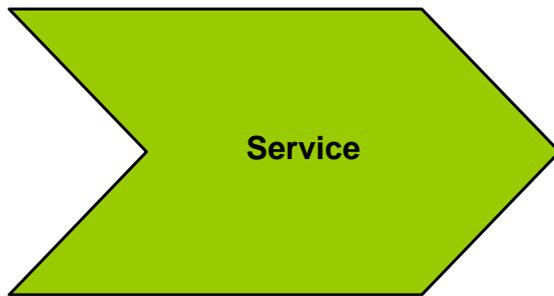
1233 • **callback (0..1) / binding : Binding (1..n)** - A service element has an optional **callback**  
1234 element used if the interface has a callback defined,, which has one or more **binding**  
1235 elements as children. The **callback** and its binding child elements are specified if there is  
1236 a need to have binding details used to handle callbacks. If the callback element is not  
1237 present, the behaviour is runtime implementation dependent.

1238

### 1239 6.1.1 Service Examples

1240

1241 The following figure shows the service symbol that used to represent a service in an assembly  
1242 diagram:

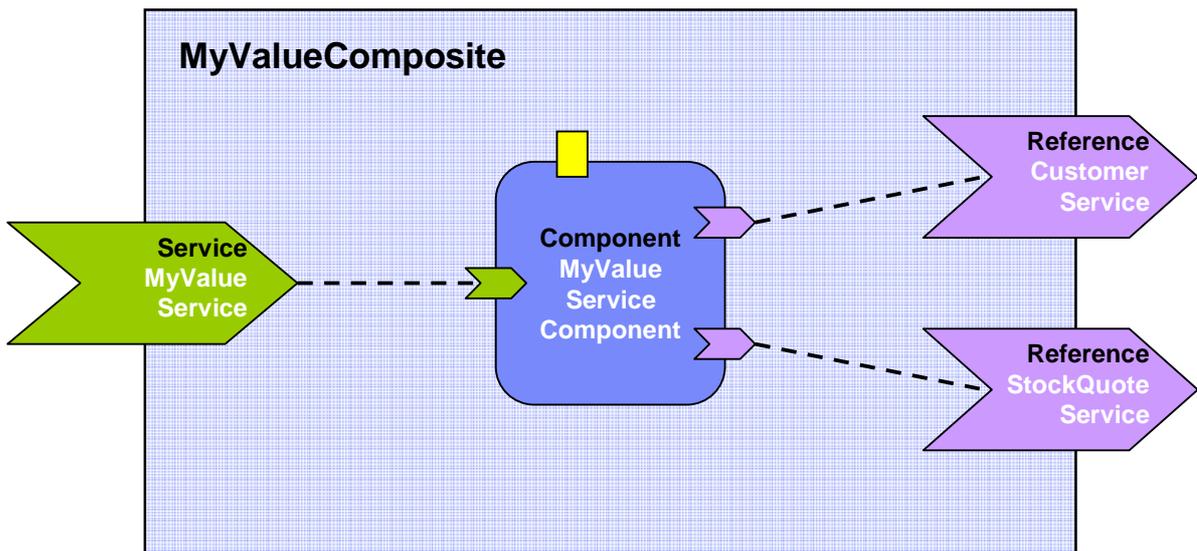


1243

1244 *Figure 7: Service symbol*

1245

1246 The following figure shows the assembly diagram for the MyValueComposite containing the service  
1247 MyValueService.



1248

1249 *Figure 8: MyValueComposite showing Service*

1250

1251 The following snippet shows the MyValueComposite.composite file for the MyValueComposite  
1252 containing the service element for the MyValueService, which is a promote of the service offered  
1253 by the MyValueServiceComponent. The name of the promoted service is omitted since  
1254 MyValueServiceComponent offers only one service. The composite service MyValueService is  
1255 bound using a Web service binding.

1256

1257

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

1258

```
<!-- MyValueComposite_4 example -->
```

1259

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

1260

```
targetNamespace="http://foo.com"
```

1261

```
name="MyValueComposite" >
```

1262

1263

```
...
```

1264

1265

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

1266

```
<interface.java interface="services.myvalue.MyValueService"/>
```

1267

```
<binding.ws port="http://www.myvalue.org/MyValueService#
```

1268

```
wsdl.endpoint(MyValueService/MyValueServiceSOAP)"/>
```

1269

```
</service>
```

1270

1271

```
<component name="MyValueServiceComponent">
```

1272

```
<implementation.java
```

1273

```
class="services.myvalue.MyValueServiceImpl"/>
```

1274

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

1275

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

1276

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

1277

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

1278

```
</component>
```

1279

1280

```
...
```

1281

1282

```
</composite>
```

1283

## 1284 6.2 Reference

1285

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**.

1289

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.

1292

1293

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

```

1294 <!-- Composite Reference schema snippet -->
1295 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
1296   ...
1297   <reference name="xs:NCName" target="list of xs:anyURI"?
1298       promote="list of xs:anyURI" wiredByImpl="xs:boolean"?
1299       multiplicity="0..1 or 1..1 or 0..n or 1..n"?
1300       requires="list of xs:QName"? policySets="list of xs:QName"?>*
1301   <interface ... />?
1302   <binding ... />*
1303   <callback?
1304       <binding ... />+
1305   </callback>
1306 </reference>
1307   ...
1308 </composite>
1309
1310

```

1311 The **composite reference** element has the following **attributes**:

- 1312 • **name : NCName (1..1)** – the name of the reference. The name must be unique across  
1313 all the composite references in the composite. The name of the composite reference can  
1314 be different then the name of the promoted component reference.
- 1315 • **promote : anyURI (1..n)** – identifies one or more promoted component references. The  
1316 value is a list of values of the form <component-name>/<reference-name> separated by  
1317 spaces. The specification of the reference name is optional if the component has only one  
1318 reference.

1319 The same component reference maybe promoted more than once, using different  
1320 composite references, but only if the multiplicity defined on the component reference is  
1321 0..n or 1..n. The multiplicity on the composite reference can restrict accordingly.

1322 Two or more component references may be promoted by one composite reference, but  
1323 only when

- 1324 • the interfaces of the component references are the same, or if the composite  
1325 reference itself declares an interface then all the component references must have  
1326 interfaces which are compatible with the composite reference interface
- 1327 • the multiplicities of the component references are compatible, i.e one can be the  
1328 restricted form of the another, which also means that the composite reference  
1329 carries the restricted form either implicitly or explicitly
- 1330 • the intents declared on the component references must be compatible – the  
1331 intents which apply to the composite reference in this case are the union of the  
1332 required intents specified for each of the promoted component references. If any  
1333 intents contradict (eg mutually incompatible qualifiers for a particular intent) then  
1334 there is an error.

- 1335 • **requires : QName (0..n)** – a list of required policy intents. See the [Policy Framework  
1336 specification \[10\]](#) for a description of this attribute. Specified **required intents** add to or  
1337 further qualify the required intents defined for the promoted component reference.

- 1338 • **policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification  
1339 \[10\]](#) for a description of this attribute.

- 1340 • **multiplicity : 0..1|1..1|0..n|1..n (1..1)** - Defines the number of wires that can  
1341 connect the reference to target services. The multiplicity can have the following values

- 1342 ○ 0..1 – zero or one wire can have the reference as a source
- 1343 ○ 1..1 – one wire can have the reference as a source
- 1344 ○ 0..n - zero or more wires can have the reference as a source
- 1345 ○ 1..n – one or more wires can have the reference as a source

1346 The value specified for the **multiplicity** attribute has to be compatible with the multiplicity  
 1347 specified on the component reference, i.e. it has to be equal or further restrict. So a  
 1348 composite reference of multiplicity 0..1 or 1..1 can be used where the promoted  
 1349 component reference has multiplicity 0..n and 1..n respectively. However, a composite  
 1350 reference of multiplicity 0..n or 1..n cannot be used to promote a component reference of  
 1351 multiplicity 0..1 or 1..1 respectively.

- 1352 • **target : anyURI (0..n)** – a list of one or more of target service URI's, depending on  
 1353 multiplicity setting. Each value wires the reference to a service in a composite that uses  
 1354 the composite containing the reference as an implementation for one of its components. For  
 1355 more details on wiring see [the section on Wires](#).
- 1356 • **wiredByImpl : boolean (0..1)** – a boolean value, "false" by default, which indicates that  
 1357 the implementation wires this reference dynamically. If set to "true" it indicates that the  
 1358 target of the reference is set at runtime by the implementation code (eg by the code  
 1359 obtaining an endpoint reference by some means and setting this as the target of the  
 1360 reference through the use of programming interfaces defined by the relevant Client and  
 1361 Implementation specification). If "true" is set, then the reference should not be wired  
 1362 statically within a using composite, but left unwired.

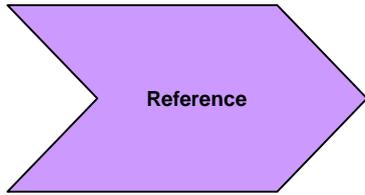
1363  
 1364 The **composite reference** element has the following **child elements**, whatever is not specified is  
 1365 defaulted from the promoted component reference(s).

- 1366 • **interface : Interface (0..1)** - If an **interface** is specified it must provide an interface  
 1367 which is the same or which is a compatible superset of the interface declared by the  
 1368 promoted component reference, i.e. provide a superset of the operations defined by the  
 1369 component for the reference. The interface is described by **zero or one interface**  
 1370 **element** which is a child element of the reference element. For details on the interface  
 1371 element see [the Interface section](#).
  - 1372 • **binding : Binding (0..n)** - If one or more **bindings** are specified they **override** any and  
 1373 all of the bindings defined for the promoted component reference from the composite  
 1374 reference perspective. The bindings defined on the component reference are still in effect  
 1375 for local wires within the composite that have the component reference as their source. A  
 1376 reference element has zero or more **binding elements** as children. Details of the binding  
 1377 element are described in the [Bindings section](#). For more details on wiring see [the section](#)  
 1378 [on Wires](#).
- 1379 A reference identifies zero or more target services which satisfy thereference. This can be  
 1380 done in a number of ways, which are fully described in section "5.3.1 Specifying the  
 1381 Target Service(s) for a Reference".
- 1382 • **callback (0..1) / binding : Binding (1..n)** - A **reference** element has an optional  
 1383 **callback** element used if the interface has a callback defined, which has one or more  
 1384 **binding** elements as children. The **callback** and its binding child elements are specified if  
 1385 there is a need to have binding details used to handle callbacks. If the callback element is  
 1386 not present, the behaviour is runtime implementation dependent.

## 1387

### 1388 6.2.1 Example Reference

1389  
 1390 The following figure shows the reference symbol that is used to represent a reference in an  
 1391 assembly diagram.



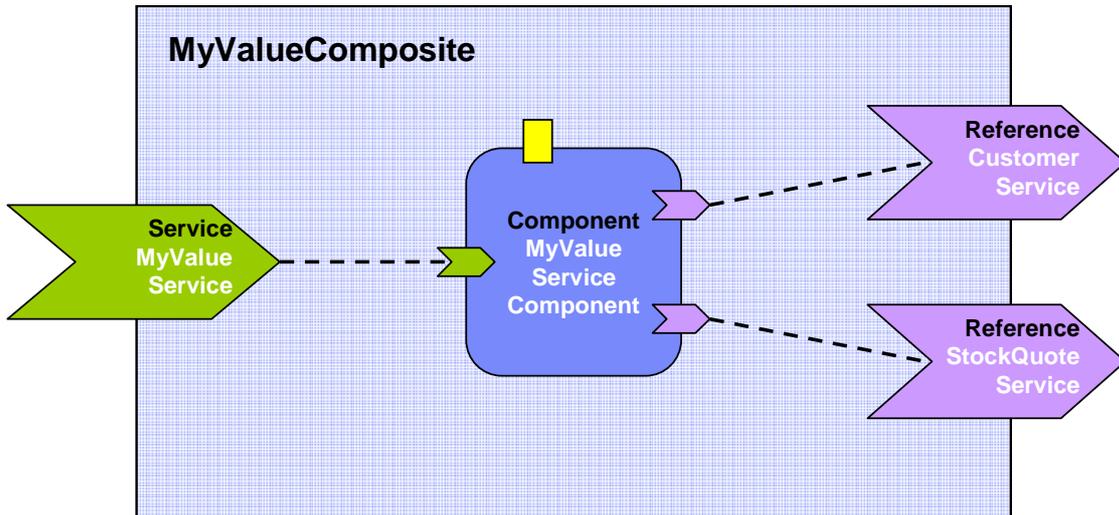
1392

1393 *Figure 9: Reference symbol*

1394

1395 The following figure shows the assembly diagram for the MyValueComposite containing the  
 1396 reference CustomerService and the reference StockQuoteService.

1397



1398

1399 *Figure 10: MyValueComposite showing References*

1400

1401 The following snippet shows the MyValueComposite.composite file for the MyValueComposite  
 1402 containing the reference elements for the CustomerService and the StockQuoteService. The  
 1403 reference CustomerService is bound using the SCA binding. The reference StockQuoteService is  
 1404 bound using the Web service binding. The endpoint addresses of the bindings can be specified,  
 1405 for example using the binding *uri* attribute (for details see the [Bindings](#) section), or  
 1406 overridden in an enclosing composite. Although in this case the reference StockQuoteService  
 1407 is bound to a Web service, its interface is defined by a Java interface, which was created  
 1408 from the WSDL portType of the target web service.

1409

```

1410 <?xml version="1.0" encoding="ASCII"?>
1411 <!-- MyValueComposite_3 example -->
1412 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1413           targetNamespace="http://foo.com"
1414           name="MyValueComposite" >
1415
1416     ...
1417
1418     <component name="MyValueServiceComponent">
  
```

```

1419         <implementation.java
1420 class="services.myvalue.MyValueServiceImpl"/>
1421         <property name="currency">EURO</property>
1422         <reference name="customerService"/>
1423         <reference name="StockQuoteService"/>
1424     </component>
1425
1426     <reference name="CustomerService"
1427         promote="MyValueServiceComponent/customerService">
1428         <interface.java interface="services.customer.CustomerService"/>
1429         <!-- The following forces the binding to be binding.sca whatever
1430 is -->
1431         <!-- specified by the component reference or by the underlying
1432 -->
1433         <!-- implementation
1434 -->
1435         <binding.sca/>
1436     </reference>
1437
1438     <reference name="StockQuoteService"
1439         promote="MyValueServiceComponent/StockQuoteService">
1440         <interface.java
1441 interface="services.stockquote.StockQuoteService"/>
1442         <binding.ws port="http://www.stockquote.org/StockQuoteService#
1443 wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
1444     </reference>
1445
1446     ...
1447
1448
1449 </composite>
1450

```

## 1451 6.3 Property

1452 **Properties** allow for the configuration of an implementation with externally set data values. A  
1453 composite can declare zero or more properties. Each property has a type, which may be either  
1454 simple or complex. An implementation may also define a default value for a property. Properties  
1455 are configured with values in the components that use the implementation.

1456 The declaration of a property in a composite follows the form described in the following schema  
1457 snippet:

```

1458
1459 <?xml version="1.0" encoding="ASCII"?>
1460 <!-- Composite Property schema snippet -->
1461 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
1462     ...
1463     <property name="xs:NCName" (type="xs:QName" | element="xs:QName")

```

```

1464         many="xs:boolean"? mustSupply="xs:boolean"?>*
1465         default-property-value?
1466     </property>
1467     ...
1468 </composite>
1469

```

1470 The **composite property** element has the following **attributes**:

- 1471     ▪ **name : NCName (1..1)** - the name of the property
- 1472     ▪ one of **(1..1)**:
  - 1473         ○ **type : QName** – the type of the property - the qualified name of an XML schema
  - 1474         type
  - 1475         ○ **element : QName** – the type of the property defined as the qualified name of an
  - 1476         XML schema global element – the type is the type of the global element
- 1477     ▪ **many : boolean (0..1)** - whether the property is single-valued (false) or multi-valued
- 1478         (true). The default is **false**. In the case of a multi-valued property, it is presented to the
- 1479         implementation as a collection of property values.
- 1480     ▪ **mustSupply : boolean (0..1)** - whether the property value must be supplied by the
- 1481         component that uses the implementation – when mustSupply="true" the component must
- 1482         supply a value since the implementation has no default value for the property. A default-
- 1483         property-value should only be supplied when mustSupply="false" (the default setting for
- 1484         the mustSupply attribute), since the implication of a default value is that it is used only
- 1485         when a value is not supplied by the using component.

1486

1487 The property element may contain an optional **default-property-value**, which provides default

1488 value for the property. The default value must match the type declared for the property:

- 1489     ○ a string, if **type** is a simple type (must match the **type** declared)
- 1490     ○ a complex type value matching the type declared by **type**
- 1491     ○ an element matching the element named by **element**
- 1492     ○ multiple values are permitted if many="true" is specified

1493

1494 Implementation types other than **composite** can declare properties in an implementation-

1495 dependent form (eg annotations within a Java class), or through a property declaration of exactly

1496 the form described above in a componentType file.

1497 Property values can be configured when an implementation is used by a component. The form of

1498 the property configuration is shown in [the section on Components](#).

### 1499 6.3.1 Property Examples

1500

1501 For the following example of Property declaration and value setting, the following complex type is

1502 used as an example:

```

1503 <xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
1504             targetNamespace="http://foo.com/"
1505             xmlns:tns="http://foo.com/">
1506     <!-- ComplexProperty schema -->
1507     <xsd:element name="fooElement" type="MyComplexType"/>
1508     <xsd:complexType name="MyComplexType">

```

```

1509         <xsd:sequence>
1510             <xsd:element name="a" type="xsd:string" />
1511             <xsd:element name="b" type="anyURI" />
1512         </xsd:sequence>
1513         <attribute name="attr" type="xsd:string" use="optional" />
1514     </xsd:complexType>
1515 </xsd:schema>
1516

```

1517 The following composite demonstrates the declaration of a property of a complex type, with a  
1518 default value, plus it demonstrates the setting of a property value of a complex type within a  
1519 component:

```

1520 <?xml version="1.0" encoding="ASCII"?>
1521
1522 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1523           xmlns:foo="http://foo.com"
1524           targetNamespace="http://foo.com"
1525           name="AccountServices">
1526 <!-- AccountServices Example1 -->
1527
1528     ...
1529
1530     <property name="complexFoo" type="foo:MyComplexType">
1531         <MyComplexPropertyValue xsi:type="foo:MyComplexType">
1532             <foo:a>AValue</foo:a>
1533             <foo:b>InterestingURI</foo:b>
1534         </MyComplexPropertyValue>
1535     </property>
1536
1537     <component name="AccountServiceComponent">
1538         <implementation.java class="foo.AccountServiceImpl" />
1539         <property name="complexBar" source="$complexFoo" />
1540         <reference name="accountDataService"
1541                 target="AccountDataServiceComponent" />
1542         <reference name="stockQuoteService" target="StockQuoteService" />
1543     </component>
1544
1545     ...
1546
1547 </composite>

```

1548 In the declaration of the property named **complexFoo** in the composite **AccountServices**, the  
1549 property is defined to be of type **foo:MyComplexType**. The namespace **foo** is declared in the  
1550 composite and it references the example XSD, where MyComplexType is defined. The declaration  
1551 of complexFoo contains a default value. This is declared as the content of the property element.  
1552 In this example, the default value consists of the element **MyComplexPropertyValue** of type

1553 foo:MyComplexType and its two child elements <foo:a> and <foo:b>, following the definition of  
1554 MyComplexType.

1555 In the component **AccountServiceComponent**, the component sets the value of the property  
1556 **complexBar**, declared by the implementation configured by the component. In this case, the  
1557 type of complexBar is foo:MyComplexType. The example shows that the value of the complexBar  
1558 property is set from the value of the complexFoo property – the **source** attribute of the property  
1559 element for complexBar declares that the value of the property is set from the value of a property  
1560 of the containing composite. The value of the source attribute is **\$complexFoo**, where  
1561 complexFoo is the name of a property of the composite. This value implies that the whole of the  
1562 value of the source property is used to set the value of the component property.

1563 The following example illustrates the setting of the value of a property of a simple type (a string)  
1564 from **part** of the value of a property of the containing composite which has a complex type:

```
1565 <?xml version="1.0" encoding="ASCII"?>
1566
1567 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1568           xmlns:foo="http://foo.com"
1569           targetNamespace="http://foo.com"
1570           name="AccountServices">
1571 <!-- AccountServices Example2 -->
1572
1573     ...
1574
1575     <property name="complexFoo" type="foo:MyComplexType">
1576         <MyComplexPropertyValue xsi:type="foo:MyComplexType">
1577             <foo:a>AValue</foo:a>
1578             <foo:b>InterestingURI</foo:b>
1579         </MyComplexPropertyValue>
1580     </property>
1581
1582     <component name="AccountServiceComponent">
1583         <implementation.java class="foo.AccountServiceImpl"/>
1584         <property name="currency" source="$complexFoo/a"/>
1585         <reference name="accountDataService"
1586             target="AccountDataServiceComponent"/>
1587         <reference name="stockQuoteService" target="StockQuoteService"/>
1588     </component>
1589
1590     ...
1591
1592 </composite>
```

1593 In this example, the component **AccountServiceComponent** sets the value of a property called  
1594 **currency**, which is of type string. The value is set from a property of the composite  
1595 **AccountServices** using the source attribute set to **\$complexFoo/a**. This is an XPath expression  
1596 that selects the property name **complexFoo** and then selects the value of the **a** subelement of  
1597 complexFoo. The "a" subelement is a string, matching the type of the currency property.

1598 Further examples of declaring properties and setting property values in a component follow:

1599 Declaration of a property with a simple type and a default value:

```
1600 <property name="SimpleTypeProperty" type="xsd:string">  
1601 MyValue  
1602 </property>
```

1603

1604 Declaration of a property with a complex type and a default value:

```
1605 <property name="complexFoo" type="foo:MyComplexType">  
1606   <MyComplexPropertyValue xsi:type="foo:MyComplexType">  
1607     <foo:a>AValue</foo:a>  
1608     <foo:b>InterestingURI</foo:b>  
1609   </MyComplexPropertyValue>  
1610 </property>
```

1611

1612 Declaration of a property with an element type:

```
1613 <property name="elementFoo" element="foo:fooElement">  
1614   <foo:fooElement>  
1615     <foo:a>AValue</foo:a>  
1616     <foo:b>InterestingURI</foo:b>  
1617   </foo:fooElement>  
1618 </property>
```

1619

1620 Property value for a simple type:

```
1621 <property name="SimpleTypeProperty">  
1622 MyValue  
1623 </property>
```

1624

1625

1626 Property value for a complex type, also showing the setting of an attribute value of the complex  
1627 type:

```
1628 <property name="complexFoo">  
1629   <MyComplexPropertyValue xsi:type="foo:MyComplexType" attr="bar">  
1630     <foo:a>AValue</foo:a>  
1631     <foo:b>InterestingURI</foo:b>  
1632   </MyComplexPropertyValue>  
1633 </property>
```

1634

1635 Property value for an element type:

```
1636 <property name="elementFoo">  
1637   <foo:fooElement attr="bar">  
1638     <foo:a>AValue</foo:a>  
1639     <foo:b>InterestingURI</foo:b>  
1640   </foo:fooElement>  
1641 </property>
```

1642

1643 Declaration of a property with a complex type where multiple values are supported:

```
1644 <property name="complexFoo" type="foo:MyComplexType" many="true"/>
```

1645

1646 Setting of a value for that property where multiple values are supplied:

```
1647 <property name="complexFoo">
```

```
1648   <MyComplexPropertyValue1 xsi:type="foo:MyComplexType" attr="bar">
```

```
1649     <foo:a>AValue</foo:a>
```

```
1650     <foo:b>InterestingURI</foo:b>
```

```
1651   </MyComplexPropertyValue1>
```

```
1652   <MyComplexPropertyValue2 xsi:type="foo:MyComplexType" attr="zing">
```

```
1653     <foo:a>BValue</foo:a>
```

```
1654     <foo:b>BoringURI</foo:b>
```

```
1655   </MyComplexPropertyValue2>
```

```
1656 </property>
```

1657

## 1658 6.4 Wire

1659 **SCA wires** within a composite connect **source component references** to **target component**  
1660 **services**.

1661 One way of defining a wire is by **configuring a reference of a component using its target**  
1662 **attribute**. The reference element is configured with the wire-target-URI of the service(s) that  
1663 resolve the reference. Multiple target services are valid when the reference has a multiplicity of  
1664 0..n or 1..n.

1665 An alternative way of defining a Wire is by means of a **wire element** which is a child of the  
1666 composite element. There can be **zero or more** wire elements in a composite. This alternative  
1667 method for defining wires is useful in circumstances where separation of the wiring from the  
1668 elements the wires connect helps simplify development or operational activities. An example is  
1669 where the components used to build a domain are relatively static but where new or changed  
1670 applications are created regularly from those components, through the creation of new assemblies  
1671 with different wiring. Deploying the wiring separately from the components allows the wiring to  
1672 be created or modified with minimum effort.

1673 Note that a Wire specified via a wire element is equivalent to a wire specified via the target  
1674 attribute of a reference. The rule which forbids mixing of wires specified with the target attribute  
1675 with the specification of endpoints in binding subelements of the reference also applies to wires  
1676 specified via separate wire elements.

1677 The following snippet shows the composite schema with the schema for the reference elements of  
1678 components and composite services and the wire child element:

1679

```
1680 <?xml version="1.0" encoding="ASCII"?>
```

```
1681 <!-- Wires schema snippet -->
```

```
1682 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
```

```
1683   targetNamespace="xs:anyURI"
```

```
1684   name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?
```

```
1685   constrainingType="QName"?
```

```
1686   requires="list of xs:QName"? policySets="list of
```

```
1687   xs:QName"?>
```

1688  
1689     ...  
1690  
1691     <wire source="xs:anyURI" target="xs:anyURI" />\*  
1692  
1693 </composite>

1696 The **reference element of a component** and the **reference element of a service** has a list of  
1697 one or more of the following **wire-target-URI** values for the target, with multiple values  
1698 separated by a space:

- 1699     • <component-name>/<service-name>
  - 1700         ○ where the target is a service of a component. The specification of the service
  - 1701         name is optional if the target component only has one service with a compatible
  - 1702         interface

1703  
1704 The **wire element** has the following attributes:

- 1705     • **source (required)** – names the source component reference. Valid URI schemes are:
  - 1706         ○ <component-name>/<reference-name>
    - 1707             ▪ where the source is a component reference. The specification of the
    - 1708             reference name is optional if the source component only has one reference
- 1709     • **target (required)** – names the target component service. Valid URI schemes are
  - 1710         ○ <component-name>/<service-name>
    - 1711             ▪ where the target is a service of a component. The specification of the
    - 1712             service name is optional if the target component only has one service with
    - 1713             a compatible interface

1714 For a composite used as a component implementation, wires can only link sources and targets  
1715 that are contained in the same composite (irrespective of which file or files are used to describe  
1716 the composite). Wiring to entities outside the composite is done through services and references  
1717 of the composite with wiring defined by the next higher composite.

1718 A wire may only connect a source to a target if the target implements an interface that is  
1719 compatible with the interface required by the source. The source and the target are compatible if:

- 1720     1. the source interface and the target interface MUST either both be remotable or they are  
1721         both local
- 1722     2. the operations on the target interface MUST be the same as or be a superset of the  
1723         operations in the interface specified on the source
- 1724     3. compatibility for the individual operation is defined as compatibility of the signature, that  
1725         is operation name, input types, and output types MUST BE the same.
- 1726     4. the order of the input and output types also MUST BE the same.
- 1727     5. the set of Faults and Exceptions expected by the source MUST BE the same or be a  
1728         superset of those specified by the target.
- 1729     6. other specified attributes of the two interfaces MUST match, including Scope and Callback  
1730         interface

1731 A Wire can connect between different interface languages (eg. Java interfaces and WSDL  
1732 portTypes) in either direction, as long as the operations defined by the two interface types are  
1733 equivalent. They are equivalent if the operation(s), parameter(s), return value(s) and  
1734 faults/exceptions map to each other.

1735 Service clients cannot (portably) ask questions at runtime about additional interfaces that are  
 1736 provided by the implementation of the service (e.g. the result of "instance of" in Java is non  
 1737 portable). It is valid for an SCA implementation to have proxies for all wires, so that, for example,  
 1738 a reference object passed to an implementation may only have the business interface of the  
 1739 reference and may not be an instance of the (Java) class which is used to implement the target  
 1740 service, even where the interface is local and the target service is running in the same process.

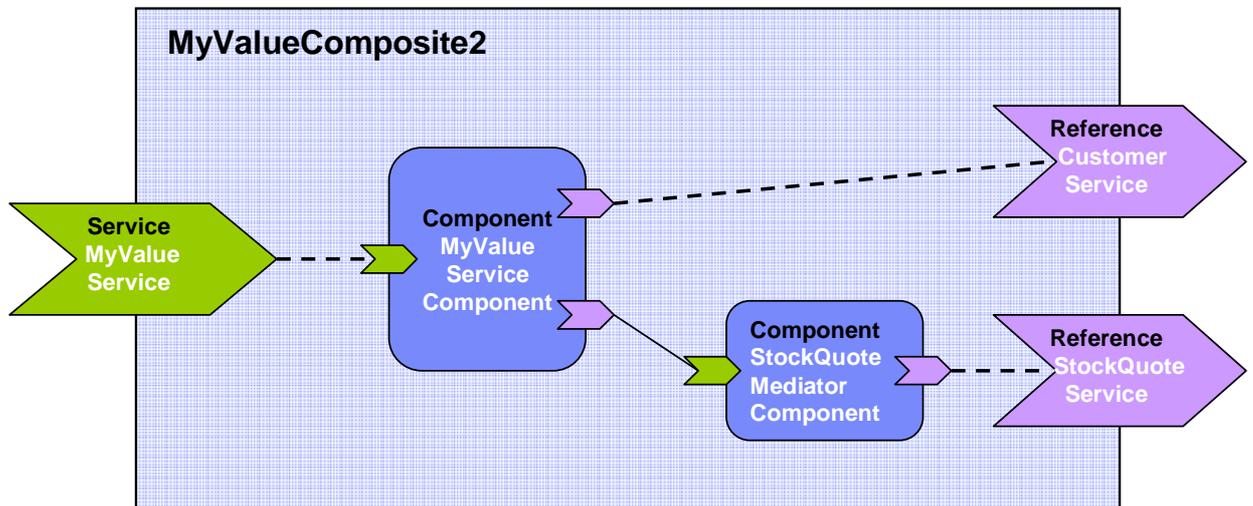
1741 **Note:** It is permitted to deploy a composite that has references that are not wired. For the case of  
 1742 an un-wired reference with multiplicity 1..1 or 1..n the deployment process provided by an SCA  
 1743 runtime SHOULD issue a warning.

1744

## 1745 6.4.1 Wire Examples

1746

1747 The following figure shows the assembly diagram for the MyValueComposite2 containing wires  
 1748 between service, components and references.



1749

1750 *Figure 11: MyValueComposite2 showing Wires*

1751

1752 The following snippet shows the MyValueComposite2.composite file for the MyValueComposite2  
 1753 containing the configured component and service references. The service MyValueService is wired  
 1754 to the MyValueServiceComponent, using an explicit <wire/> element. The  
 1755 MyValueServiceComponent's customerService reference is wired to the composite's  
 1756 CustomerService reference. The MyValueServiceComponent's stockQuoteService reference is  
 1757 wired to the StockQuoteMediatorComponent, which in turn has its reference wired to the  
 1758 StockQuoteService reference of the composite.

1759

```
1760 <?xml version="1.0" encoding="ASCII"?>
1761 <!-- MyValueComposite Wires examples -->
1762 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1763           targetNamespace="http://foo.com"
1764           name="MyValueComposite2" >
1765
1766     <service name="MyValueService" promote="MyValueServiceComponent">
1767       <interface.java interface="services.myvalue.MyValueService"/>
```

```

1768         <binding.ws port="http://www.myvalue.org/MyValueService#
1769             wsdl.endpoint(MyValueService/MyValueServiceSOAP)"/>
1770     </service>
1771
1772     <component name="MyValueServiceComponent">
1773         <implementation.java
1774             class="services.myvalue.MyValueServiceImpl"/>
1775         <property name="currency">EURO</property>
1776         <service name="MyValueService"/>
1777         <reference name="customerService"/>
1778         <reference name="stockQuoteService"/>
1779     </component>
1780
1781     <wire source="MyValueServiceComponent/stockQuoteService"
1782         target="StockQuoteMediatorComponent"/>
1783
1784     <component name="StockQuoteMediatorComponent">
1785         <implementation.java class="services.myvalue.SQMediatorImpl"/>
1786         <property name="currency">EURO</property>
1787         <reference name="stockQuoteService"/>
1788     </component>
1789
1790     <reference name="CustomerService"
1791         promote="MyValueServiceComponent/customerService">
1792         <interface.java interface="services.customer.CustomerService"/>
1793         <binding.sca/>
1794     </reference>
1795
1796     <reference name="StockQuoteService"
1797         promote="StockQuoteMediatorComponent">
1798         <interface.java
1799             interface="services.stockquote.StockQuoteService"/>
1800         <binding.ws port="http://www.stockquote.org/StockQuoteService#
1801             wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
1802     </reference>
1803
1804 </composite>
1805

```

## 1806 6.4.2 Autowire

1807 SCA provides a feature named **Autowire**, which can help to simplify the assembly of composites.  
1808 Autowire enables component references to be automatically wired to component services which  
1809 will satisfy those references, without the need to create explicit wires between the references and  
1810 the services. When the autowire feature is used, a component reference which is not promoted  
1811 and which is not explicitly wired to a service within a composite is automatically wired to a target

1812 service within the same composite. Autowire works by searching within the composite for a  
1813 service interface which matches the interface of the references.

1814 The autowire feature is not used by default. Autowire is enabled by the setting of an autowire  
1815 attribute to "true". Autowire is disabled by setting of the autowire attribute to "false" The autowire  
1816 attribute can be applied to any of the following elements within a composite:

- 1817 • reference
- 1818 • component
- 1819 • composite

1820 Where an element does not have an explicit setting for the autowire attribute, it inherits the  
1821 setting from its parent element. Thus a reference element inherits the setting from its containing  
1822 component. A component element inherits the setting from its containing composite. Where  
1823 there is no setting on any level, autowire="false" is the default.

1824 As an example, if a composite element has autowire="true" set, this means that autowiring is  
1825 enabled for all component references within that composite. In this example, autowiring can be  
1826 turned off for specific components and specific references through setting autowire="false" on the  
1827 components and references concerned.

1828 For each component reference for which autowire is enabled, the autowire process searches within  
1829 the composite for target services which are compatible with the reference. "Compatible" here  
1830 means:

- 1831 • the target service interface must be a compatible superset of the reference interface (as  
1832 defined in [the section on Wires](#))
- 1833 • the intents, and policies applied to the service must be compatible on the reference – so  
1834 that wiring the reference to the service will not cause an error due to policy mismatch (see  
1835 [the Policy Framework specification \[10\]](#) for details)

1836 If the search finds **more than 1** valid target service for a particular reference, the action taken  
1837 depends on the multiplicity of the reference:

- 1838 • for multiplicity 0..1 and 1..1, the SCA runtime selects one of the target services in a  
1839 runtime-dependent fashion and wires the reference to that target service
- 1840 • for multiplicity 0..n and 1..n, the reference is wired to all of the target services

1841 If the search finds **no** valid target services for a particular reference, the action taken depends on  
1842 the multiplicity of the reference:

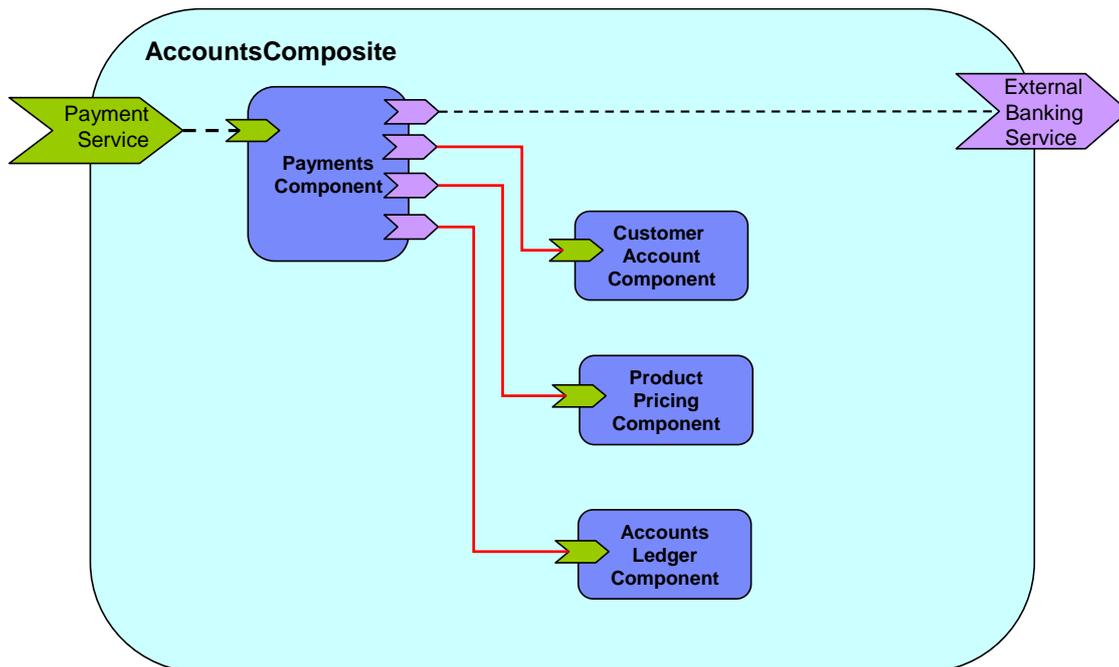
- 1843 • for multiplicity 0..1 and 0..n, there is no problem – no services are wired and there is no  
1844 error
- 1845 • for multiplicity 1..1 and 1..n, an error is raised by the SCA runtime since the reference is  
1846 intended to be wired

1847

### 1848 6.4.3 Autowire Examples

1849 This example demonstrates two versions of the same composite – the first version is done using  
1850 explicit wires, with no autowiring used, the second version is done using autowire. In both cases  
1851 the end result is the same – the same wires connect the references to the services.

1852 First, here is a diagram for the composite:



1853

1854 *Figure 12: Example Composite for Autowire*

1855 First, the composite using explicit wires:

1856

```

1856 <?xml version="1.0" encoding="UTF-8"?>
1857 <!-- Autowire Example - No autowire -->
1858 <composite xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
1859           xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1860           xmlns:foo="http://foo.com"
1861           targetNamespace="http://foo.com"
1862           name="AccountComposite">
1863
1864   <service name="PaymentService" promote="PaymentsComponent"/>
1865
1866   <component name="PaymentsComponent">
1867     <implementation.java class="com.foo.accounts.Payments"/>
1868     <service name="PaymentService"/>
1869     <reference name="CustomerAccountService"
1870               target="CustomerAccountComponent"/>
1871     <reference name="ProductPricingService"
1872               target="ProductPricingComponent"/>
1873     <reference name="AccountsLedgerService"
1874               target="AccountsLedgerComponent"/>
1875     <reference name="ExternalBankingService"/>
1876   </component>
1877
1878   <component name="CustomerAccountComponent">

```

```

1879         <implementation.java class="com.foo.accounts.CustomerAccount" />
1880     </component>
1881
1882     <component name="ProductPricingComponent">
1883         <implementation.java class="com.foo.accounts.ProductPricing" />
1884     </component>
1885
1886     <component name="AccountsLedgerComponent">
1887         <implementation.composite name="foo:AccountsLedgerComposite" />
1888     </component>
1889
1890     <reference name="ExternalBankingService"
1891         promote="PaymentsComponent/ExternalBankingService" />
1892
1893 </composite>
1894
1895 Secondly, the composite using autowire:
1896 <?xml version="1.0" encoding="UTF-8"?>
1897 <!-- Autowire Example - With autowire -->
1898 <composite xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
1899     xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1900     xmlns:foo="http://foo.com"
1901     targetNamespace="http://foo.com"
1902     name="AccountComposite">
1903
1904     <service name="PaymentService" promote="PaymentsComponent">
1905         <interface.java class="com.foo.PaymentServiceInterface" />
1906     </service>
1907
1908     <component name="PaymentsComponent" autowire="true">
1909         <implementation.java class="com.foo.accounts.Payments" />
1910         <service name="PaymentService" />
1911         <reference name="CustomerAccountService" />
1912         <reference name="ProductPricingService" />
1913         <reference name="AccountsLedgerService" />
1914         <reference name="ExternalBankingService" />
1915     </component>
1916
1917     <component name="CustomerAccountComponent">
1918         <implementation.java class="com.foo.accounts.CustomerAccount" />
1919     </component>
1920
1921     <component name="ProductPricingComponent">

```

```

1922         <implementation.java class="com.foo.accounts.ProductPricing"/>
1923     </component>
1924
1925     <component name="AccountsLedgerComponent">
1926         <implementation.composite name="foo:AccountsLedgerComposite"/>
1927     </component>
1928
1929     <reference name="ExternalBankingService"
1930         promote="PaymentsComponent/ExternalBankingService"/>
1931
1932 </composite>

```

1933 In this second case, autowire is set on for the PaymentsComponent and there are no explicit wires  
1934 for any of its references – the wires are created automatically through autowire.

1935 **Note:** In the second example, it would be possible to omit all of the service and reference  
1936 elements from the PaymentsComponent. They are left in for clarity, but if they are omitted, the  
1937 component service and references still exist, since they are provided by the implementation used  
1938 by the component.

1939

## 1940 6.5 Using Composites as Component Implementations

1941 Composites may form **component implementations** in higher-level composites – in other words  
1942 the higher-level composites can have components which are implemented by composites.

1943 When a composite is used as a component implementation, it defines a boundary of visibility.  
1944 Components within the composite cannot be referenced directly by the using component. The  
1945 using component can only connect wires to the services and references of the used composite and  
1946 set values for any properties of the composite. The internal construction of the composite is  
1947 invisible to the using component.

1948 A composite used as a component implementation must also honor a **completeness contract**.  
1949 The services, references and properties of the composite form a contract which is relied upon by  
1950 the using component. The concept of completeness of the composite implies:

- 1951 • the composite must have at least one service or at least one reference.  
1952 A component with no services and no references is not meaningful in terms of SCA, since  
1953 it cannot be wired to anything – it neither provides nor consumes any services  
1954
- 1955 • each service offered by the composite must be wired to a service of a component or to a  
1956 composite reference.  
1957 If services are left unwired, the implication is that some exception will occur at runtime if  
1958 the service is invoked.

1959 The component type of a composite is defined by the set of service elements, reference elements  
1960 and property elements that are the children of the composite element.

1961 Composites are used as component implementations through the use of the  
1962 **implementation.composite** element as a child element of the component. The schema snippet  
1963 for the implementation.composite element is:

```

1964
1965 <?xml version="1.0" encoding="ASCII"?>
1966 <!-- Composite Implementation schema snippet -->
1967 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1968     targetNamespace="xs:anyURI"

```

```

1969         name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?
1970         constrainingType="QName"?
1971         requires="list of xs:QName"? policySets="list of
1972 xs:QName"?>
1973
1974     ...
1975
1976     <component name="xs:NCName" autowire="xs:boolean"?
1977         requires="list of xs:QName"? policySets="list of xs:QName"?>*
1978         <implementation.composite name="xs:QName"/>?
1979         <service name="xs:NCName" requires="list of xs:QName"?
1980             policySets="list of xs:QName"?>*
1981             <interface ... />?
1982             <binding uri="xs:anyURI" name="xs:QName"?
1983                 requires="list of xs:QName"
1984                 policySets="list of xs:QName"?/>*
1985             <callback>?
1986                 <binding uri="xs:anyURI"? name="xs:QName"?
1987                     requires="list of xs:QName"?
1988                     policySets="list of xs:QName"?/>+
1989             </callback>
1990         </service>
1991         <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
1992             source="xs:string"? file="xs:anyURI"?>*
1993             property-value
1994         </property>
1995         <reference name="xs:NCName" target="list of xs:anyURI"?
1996             autowire="xs:boolean"? wiredByImpl="xs:boolean"?
1997             requires="list of xs:QName"? policySets="list of xs:QName"?
1998             multiplicity="0..1 or 1..1 or 0..n or 1..n"?/>*
1999             <interface ... />?
2000             <binding uri="xs:anyURI"? name="xs:QName"?
2001                 requires="list of xs:QName" policySets="list of
2002 xs:QName"?/>*
2003             <callback>?
2004                 <binding uri="xs:anyURI"? name="xs:QName"?
2005                     requires="list of xs:QName"?
2006                     policySets="list of xs:QName"?/>+
2007             </callback>
2008             </reference>
2009         </component>
2010
2011     ...

```

2012  
2013  
2014  
2015  
2016  
2017  
2018  
2019  
2020  
2021  
2022  
2023  
2024  
2025  
2026  
2027  
2028  
2029  
2030  
2031  
2032  
2033  
2034  
2035  
2036  
2037  
2038  
2039  
2040  
2041  
2042  
2043  
2044  
2045  
2046  
2047  
2048  
2049  
2050  
2051  
2052  
2053  
2054

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

```

2055         <reference name="StockQuoteService"/>
2056
2057         <property name="currency" source="$currency"/>
2058     </component>
2059
2060     <component name="AccountDataService">
2061         <implementation.composite name="foo:AccountDataServiceComposite"/>
2062
2063         <property name="currency" source="$currency"/>
2064     </component>
2065
2066 </composite>
2067

```

## 2068 6.6 Using Composites through Inclusion

2069 In order to assist team development, composites may be developed in the form of multiple  
 2070 physical artifacts that are merged into a single logical unit.

2071 A composite is defined in an **xxx.composite** file and the composite may receive additional  
 2072 content through the **inclusion of other composite** files.

2073 The semantics of included composites are that the content of the included composite is inlined into  
 2074 the using composite **xxx.composite** file through **include** elements in the using composite. The  
 2075 effect is one of **textual inclusion** – that is, the text content of the included composite is placed  
 2076 into the using composite in place of the include statement. The included composite element itself  
 2077 is discarded in this process – only its contents are included.

2078 The composite file used for inclusion can have any contents, but always contains a single  
 2079 **composite** element. The composite element may contain any of the elements which are valid as  
 2080 child elements of a composite element, namely components, services, references, wires and  
 2081 includes. There is no need for the content of an included composite to be complete, so that  
 2082 artifacts defined within the using composite or in another associated included composite file may  
 2083 be referenced. For example, it is permissible to have two components in one composite file while a  
 2084 wire specifying one component as the source and the other as the target can be defined in a  
 2085 second included composite file.

2086 It is an error if the (using) composite resulting from the inclusion is invalid – for example, if there  
 2087 are duplicated elements in the using composite (eg. two services with the same uri contributed by  
 2088 different included composites), or if there are wires with non-existent source or target.

2089 The following snippet shows the partial schema for the include element.

```

2090
2091 <?xml version="1.0" encoding="UTF-8"?>
2092 <!-- Include snippet -->
2093 <composite      xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2094                targetNamespace="xs:anyURI"
2095                name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?
2096                constrainingType="QName"?
2097                requires="list of xs:QName"? policySets="list of
2098 xs:QName"?>
2099
2100     ...

```

```

2101
2102     <include name="xs:QName" />*
2103
2104     ...
2105
2106 </composite>
2107

```

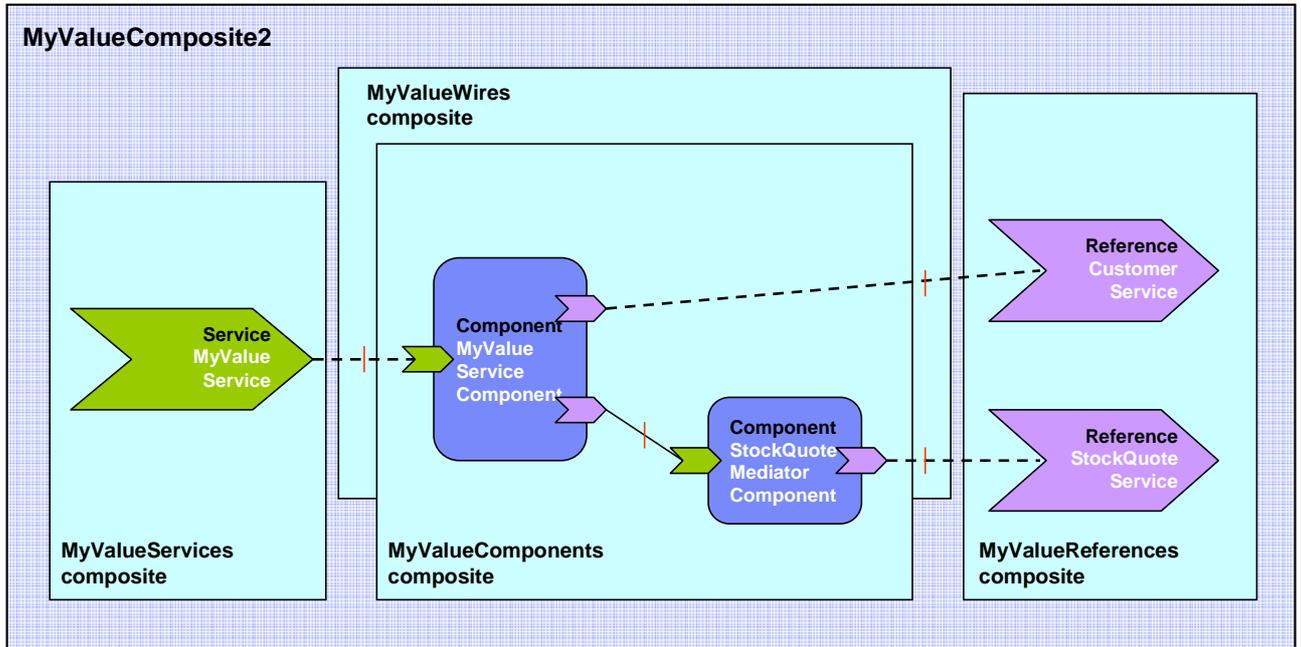
2108 The include element has the following **attribute**:

- 2109 • **name (required)** – the name of the composite that is included.
- 2110

### 2111 6.6.1 Included Composite Examples

2112

2113 The following figure shows the assembly diagram for the MyValueComposite2 containing four  
 2114 included composites. The **MyValueServices composite** contains the MyValueService service. The  
 2115 **MyValueComponents composite** contains the MyValueServiceComponent and the  
 2116 StockQuoteMediatorComponent as well as the wire between them. The **MyValueReferences**  
 2117 **composite** contains the CustomerService and StockQuoteService references. The **MyValueWires**  
 2118 **composite** contains the wires that connect the MyValueService service to the  
 2119 MyValueServiceComponent, that connect the customerService reference of the  
 2120 MyValueServiceComponent to the CustomerService reference, and that connect the  
 2121 stockQuoteService reference of the StockQuoteMediatorComponent to the StockQuoteService  
 2122 reference. Note that this is just one possible way of building the MyValueComposite2 from a set of  
 2123 included composites.



2124  
 2125  
 2126 *Figure 13 MyValueComposite2 built from 4 included composites*

2127  
 2128 The following snippet shows the contents of the MyValueComposite2.composite file for the  
 2129 MyValueComposite2 built using included composites. In this sample it only provides the name of

2130 the composite. The composite file itself could be used in a scenario using included composites to  
2131 define components, services, references and wires.

2132

```
2133 <?xml version="1.0" encoding="ASCII"?>
2134 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2135           targetNamespace="http://foo.com"
2136           xmlns:foo="http://foo.com"
2137           name="MyValueComposite2" >
2138
2139     <include name="foo:MyValueServices"/>
2140     <include name="foo:MyValueComponents"/>
2141     <include name="foo:MyValueReferences"/>
2142     <include name="foo:MyValueWires"/>
2143
2144 </composite>
```

2145

2146 The following snippet shows the content of the MyValueServices.composite file.

2147

```
2148 <?xml version="1.0" encoding="ASCII"?>
2149 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2150           targetNamespace="http://foo.com"
2151           xmlns:foo="http://foo.com"
2152           name="MyValueServices" >
2153
2154     <service name="MyValueService" promote="MyValueServiceComponent">
2155       <interface.java interface="services.myvalue.MyValueService"/>
2156       <binding.ws port="http://www.myvalue.org/MyValueService#
2157                 wsdl.endpoint(MyValueService/MyValueServiceSOAP)"/>
2158     </service>
2159
2160 </composite>
```

2161

2162 The following snippet shows the content of the MyValueComponents.composite file.

2163

```
2164 <?xml version="1.0" encoding="ASCII"?>
2165 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2166           targetNamespace="http://foo.com"
2167           xmlns:foo="http://foo.com"
2168           name="MyValueComponents" >
2169
2170     <component name="MyValueServiceComponent">
2171       <implementation.java
2172       class="services.myvalue.MyValueServiceImpl"/>
```

```

2173         <property name="currency">EURO</property>
2174     </component>
2175
2176     <component name="StockQuoteMediatorComponent">
2177         <implementation.java class="services.myvalue.SQMediatorImpl"/>
2178         <property name="currency">EURO</property>
2179     </component>
2180
2181 <composite>
2182

```

2183 The following snippet shows the content of the MyValueReferences.composite file.

```

2184
2185 <?xml version="1.0" encoding="ASCII"?>
2186 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2187           targetNamespace="http://foo.com"
2188           xmlns:foo="http://foo.com"
2189           name="MyValueReferences" >
2190
2191     <reference name="CustomerService"
2192             promote="MyValueServiceComponent/CustomerService">
2193         <interface.java interface="services.customer.CustomerService"/>
2194         <binding.sca/>
2195     </reference>
2196
2197     <reference name="StockQuoteService"
2198             promote="StockQuoteMediatorComponent">
2199         <interface.java
2200             interface="services.stockquote.StockQuoteService"/>
2201         <binding.ws port="http://www.stockquote.org/StockQuoteService#
2202             wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
2203     </reference>
2204
2205 </composite>

```

2206 The following snippet shows the content of the MyValueWires.composite file.

```

2207
2208 <?xml version="1.0" encoding="ASCII"?>
2209 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2210           targetNamespace="http://foo.com"
2211           xmlns:foo="http://foo.com"
2212           name="MyValueWires" >
2213
2214     <wire source="MyValueServiceComponent/stockQuoteService"
2215           target="StockQuoteMediatorComponent"/>

```

2216  
2217

`</composite>`

2218  
2219

## **6.7 Composites which Include Component Implementations of Multiple Types**

2220

2221  
2222  
2223  
2224

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

2225

## 7 ConstrainingType

2226  
2227  
2228  
2229  
2230

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.

2231  
2232  
2233  
2234  
2235  
2236

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.

2237  
2238  
2239  
2240  
2241  
2242

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`.

2243  
2244

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

2245  
2246

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

2247

2248

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

2249

```
<!-- ConstrainingType schema snippet -->
```

2250

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

2251

```
    targetNamespace="xs:anyURI"?
```

2252

```
    name="xs:NCName" requires="list of xs:QName"?>
```

2253

2254

2255

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

2256

```
        <interface ... />?
```

2257

```
    </service>
```

2258

2259

```
    <reference name="xs:NCName"
```

2260

```
        multiplicity="0..1 or 1..1 or 0..n or 1..n"?
```

2261

```
        requires="list of xs:QName"?>*
```

2262

```
        <interface ... />?
```

2263

```
    </reference>
```

2264

2265

```
    <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
```

2266

```
        many="xs:boolean"? mustSupply="xs:boolean"?>*
```

2267

```
        default-property-value?
```

2268

```
    </property>
```

2269

2270  
2271  
2272

```
</constrainingType>
```

2273 The constrainingType element has the following **attributes**:

- 2274 • **name (required)** – the name of the constrainingType. The form of a constrainingType  
2275 name is an XML QName, in the namespace identified by the targetNamespace attribute.
- 2276 • **targetNamespace (optional)** – an identifier for a target namespace into which the  
2277 constrainingType is declared
- 2278 • **requires (optional)** – a list of policy intents. See [the Policy Framework specification \[10\]](#)  
2279 for a description of this attribute.

2280 ConstrainingType contains **zero or more properties, services, references**.

2281

2282 When an implementation is constrained by a constrainingType it must define all the services,  
2283 references and properties specified in the corresponding constrainingType. The constraining type's  
2284 references and services will have interfaces specified and may have intents specified. An  
2285 implementation may contain additional services, additional optional references and additional  
2286 optional properties, but cannot contain additional non-optional references or additional non-  
2287 optional properties (a non-optional property is one with no default value applied).

2288 When a component is constrained by a constrainingType (via the "constrainingType" attribute),  
2289 the entire componentType associated with the component and its implementation is not visible to  
2290 the containing composite. The containing composite can only see a projection of the  
2291 componentType associated with the component and implementation as scoped by the  
2292 constrainingType of the component. For example, an additional service provided by the  
2293 implementation which is not in the constrainingType associated with the component cannot be  
2294 promoted by the containing composite. This requirement ensures that the constrainingType  
2295 contract cannot be violated by the composite.

2296 The constrainingType can include required intents on any element. Those intents are applied to  
2297 any component that uses that constrainingType. In other words, if requires="reliability" exists on  
2298 a constrainingType, or its child service or reference elements, then a constrained component or its  
2299 implementation must include requires="reliability" on the component or implementation or on its  
2300 corresponding service or reference. Note that the component or implementation may use a  
2301 qualified form of an intent specified in unqualified form in the constrainingType, but if the  
2302 constrainingType uses the qualified form, then the component or implementation must also use  
2303 the qualified form, otherwise there is an error.

2304 A constrainingType can be applied to an implementation. In this case, the implementation's  
2305 componentType has a constrainingType attribute set to the QName of the constrainingType.

2306

## 2307 7.1 Example constrainingType

2308

2309 The following snippet shows the contents of the component called "MyValueServiceComponent"  
2310 which is constrained by the constrainingType myns:CT. The componentType associated with the  
2311 implementation is also shown.

2312

```
2313 <component name="MyValueServiceComponent" constrainingType="myns:CT">  
2314   <implementation.java class="services.myvalue.MyValueServiceImpl"/>  
2315   <property name="currency">EURO</property>  
2316   <reference name="customerService" target="CustomerService">  
2317     <binding.ws ...>
```

```

2318     <reference name="StockQuoteService"
2319         target="StockQuoteMediatorComponent" />
2320 </component>
2321
2322 <constrainingType name="CT"
2323     targetNamespace="http://mysns.com">
2324     <service name="MyValueService">
2325         <interface.java interface="services.myvalue.MyValueService" />
2326     </service>
2327     <reference name="customerService">
2328         <interface.java interface="services.customer.CustomerService" />
2329     </reference>
2330     <reference name="stockQuoteService">
2331         <interface.java interface="services.stockquote.StockQuoteService" />
2332     </reference>
2333     <property name="currency" type="xsd:string" />
2334 </constrainingType>

```

2335 The component MyValueServiceComponent is constrained by the constrainingType CT which  
2336 means that it must provide:

- 2337 • service **MyValueService** with the interface services.myvalue.MyValueService
- 2338 • reference **customerService** with the interface services.stockquote.StockQuoteService
- 2339 • reference **stockQuoteService** with the interface services.stockquote.StockQuoteService
- 2340 • property **currency** of type xsd:string.

---

## 2341 8 Interface

2342 **Interfaces** define one or more business functions. These business functions are provided by  
2343 Services and are used by References. A Service offers the business functionality of exactly one  
2344 interface for use by other components. Each interface defines one or more service **operations**  
2345 and each operation has zero or one **request (input) message** and zero or one **response**  
2346 **(output) message**. The request and response messages may be simple types such as a string  
2347 value or they may be complex types.

2348 SCA currently supports the following interface type systems:

- 2349 • Java interfaces
- 2350 • WSDL 1.1 portTypes ([Web Services Definition Language \[8\]](#))
- 2351 • WSDL 2.0 interfaces ([Web Services Definition Language \[8\]](#))
- 2352 • C++ classes

2353 SCA is also extensible in terms of interface types. Support for other interface type systems can be  
2354 added through the extensibility mechanisms of SCA, as described in [the Extension Model section](#).

2355

2356 The following snippet shows the definition for the **interface** base element.

2357

```
2358 <interface requires="list of xs:QName"? policySets="list of xs:QName"?/>  
2359
```

2360 The **interface** base element has the following **attributes**:

- 2361 • **requires : QName (0..n)** – a list of policy intents. See the [Policy Framework specification](#)  
2362 [\[10\]](#) for a description of this attribute
- 2363 • **policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification](#)  
2364 [\[10\]](#) for a description of this attribute.

2365

2366 The following snippet shows a sample for the WSDL portType (WSDL 1.1) or WSDL interface  
2367 (WSDL 2.0) element.

2368

```
2369 <interface.wSDL interface="xs:anyURI" ... />
```

2370

2371 The interface.wSDL element has the following attributes:

- 2372 • **interface** – URI of the portType/interface with the following format  
2373 ○ `<WSDL-namespace-URI>#wsdl.interface(<portTypeOrInterface-name>)`

2374

2375 The following snippet shows a sample for the WSDL portType/interface element.

2376

```
2377 <interface.wSDL interface="http://www.stockquote.org/StockQuoteService#  
2378 wsdl.interface(StockQuo  
2379 te)"/>
```

2380

2381 For WSDL 1.1, the interface attribute points to a portType in the WSDL. For WSDL 2.0, the  
2382 interface attribute points to an interface in the WSDL. For the WSDL 1.1 portType and WSDL 2.0

2383 interface type systems, arguments and return of the service operations are described using XML  
2384 schema.

2385 For information about Java interfaces, including details of SCA-specific annotations, see the SCA  
2386 Java Common Annotations and APIs specification [1].

## 2387 8.1 Local and Remotable Interfaces

2388 A remotable service is one which may be called by a client which is running in an operating system  
2389 process different from that of the service itself (this also applies to clients running on different  
2390 machines from the service). Whether a service of a component implementation is remotable is  
2391 defined by the interface of the service. In the case of Java this is defined by adding the  
2392 **@Remotable** annotation to the Java interface (see [Client and Implementation Model Specification  
2393 for Java](#)). WSDL defined interfaces are always remotable.

2394

2395 The style of remotable interfaces is typically **coarse grained** and intended for **loosely coupled**  
2396 interactions. Remotable service Interfaces MUST NOT make use of **method or operation**  
2397 **overloading**.

2398

2399 Independent of whether the remotable service is called remotely from outside the process where  
2400 the service runs or from another component running in the same process, the data exchange  
2401 semantics are **by-value**.

2402 Implementations of remotable services may modify input messages (parameters) during or after  
2403 an invocation and may modify return messages (results) after the invocation. If a remotable  
2404 service is called locally or remotely, the SCA container is responsible for making sure that no  
2405 modification of input messages or post-invocation modifications to return messages are seen by  
2406 the caller.

2407 Here is a snippet which shows an example of a remotable java interface:

2408

```
2409 package services.hello;
```

2410

```
2411 @Remotable
```

```
2412 public interface HelloService {
```

2413

```
2414     String hello(String message);
```

```
2415 }
```

2416

2417 It is possible for the implementation of a remotable service to indicate that it can be called using  
2418 by-reference data exchange semantics when it is called from a component in the same process.  
2419 This can be used to improve performance for service invocations between components that run in  
2420 the same process. This can be done using the @AllowsPassByReference annotation (see the [Java  
2421 Client and Implementation Specification](#)).

2422

2423 A service typed by a local interface can only be called by clients that are running in the same  
2424 process as the component that implements the local service. Local services cannot be published  
2425 via remotable services of a containing composite. In the case of Java a local service is defined by a  
2426 Java interface definition without a **@Remotable** annotation.

2427

2428 The style of local interfaces is typically **fine grained** and intended for **tightly coupled**  
2429 interactions. Local service interfaces can make use of **method or operation overloading**.

2430 The data exchange semantic for calls to services typed by local interfaces is **by-reference**.

2431

## 2432 8.2 Bidirectional Interfaces

2433 The relationship of a business service to another business service is often peer-to-peer, requiring  
2434 a two-way dependency at the service level. In other words, a business service represents both a  
2435 consumer of a service provided by a partner business service and a provider of a service to the  
2436 partner business service. This is especially the case when the interactions are based on  
2437 asynchronous messaging rather than on remote procedure calls. The notion of **bidirectional**  
2438 **interfaces** is used in SCA to directly model peer-to-peer bidirectional business service  
2439 relationships.

2440 An interface element for a particular interface type system must allow the specification of an  
2441 optional callback interface. If a callback interface is specified SCA refers to the interface as a whole  
2442 as a bidirectional interface.

2443 The following snippet shows the interface element defined using Java interfaces with an optional  
2444 callbackInterface attribute.

2445

```
2446 <interface.java          interface="services.invoicing.ComputePrice"  
2447                       callbackInterface="services.invoicing.InvoiceCallback"/>
```

2448

2449 If a service is defined using a bidirectional interface element then its implementation implements  
2450 the interface, and its implementation uses the callback interface to converse with the client that  
2451 called the service interface.

2452

2453 If a reference is defined using a bidirectional interface element, the client component  
2454 implementation using the reference calls the referenced service using the interface. The client  
2455 must provide an implementation of the callback interface.

2456 Callbacks may be used for both remotable and local services. Either both interfaces of a  
2457 bidirectional service MUST be remotable, or both MUST be local. A bidirectional service MUST NOT  
2458 mix local and remote services.

2459

## 2460 8.3 Conversational Interfaces

2461

2462 Services sometimes cannot easily be defined so that each operation stands alone and is  
2463 completely independent of the other operations of the same service. Instead, there is a sequence  
2464 of operations that must be called in order to achieve some higher level goal. SCA calls this  
2465 sequence of operations a **conversation**. If the service uses a bidirectional interface, the  
2466 conversation may include both operations and callbacks.

2467

2468 Such conversational services are typically managed by using conversation identifiers that are  
2469 either (1) part of the application data (message parts or operation parameters) or 2)  
2470 communicated separately from application data (possibly in headers). SCA introduces the concept  
2471 of **conversational interfaces** for describing the interface contract for conversational services of the  
2472 second form above. With this form, it is possible for the runtime to automatically manage the  
2473 conversation, with the help of an appropriate binding specified at deployment. SCA does not  
2474 standardize any aspect of conversational services that are maintained using application data.  
2475 Such services are neither helped nor hindered by SCA's conversational service support.

2476

2477 Conversational services typically involve state data that relates to the conversation that is taking  
2478 place. The creation and management of the state data for a conversation has a significant impact  
2479 on the development of both clients and implementations of conversational services.

2480

2481 Traditionally, application developers who have needed to write conversational services have been  
2482 required to write a lot of plumbing code. They need to:

2483

2484 - choose or define a protocol to communicate conversational (correlation) information  
2485 between the client & provider

2486 - route conversational messages in the provider to a machine that can handle that  
2487 conversation, while handling concurrent data access issues

2488 - write code in the client to use/encode the conversational information

2489 - maintain state that is specific to the conversation, sometimes persistently and  
2490 transactionally, both in the implementation and the client.

2491

2492 SCA makes it possible to divide the effort associated with conversational services between a  
2493 number of roles:

2494 - Application Developer: Declares that a service interface is conversational (leaving the  
2495 details of the protocol up to the binding). Uses lifecycle semantics, APIs or other  
2496 programmatic mechanisms (as defined by the implementation-type being used) to  
2497 manage conversational state.

2498 - Application Assembler: chooses a binding that can support conversations

2499 - Binding Provider: implements a protocol that can pass conversational information with  
2500 each operation request/response.

2501 - Implementation-Type Provider: defines APIs and/or other programmatic mechanisms for  
2502 application developers to access conversational information. Optionally implements  
2503 instance lifecycle semantics that automatically manage implementation state based on  
2504 the binding's conversational information.

2505

2506 There is a policy intent with the name conversational which is used to mark an interface as being  
2507 conversational in nature. Where a service or a reference has a conversational interface, the  
2508 conversational intent MUST be attached either to the interface itself, or to the service or reference  
2509 using the interface. How to attach the conversational intent to an interface depends on the type of  
2510 the interface. For a WSDL interface, this is described in section 8.4 "SCA-Specific Aspects for  
2511 WSDL Interfaces". For a Java interface, it is described in the Java Common Annotations and APIs  
2512 specification. Note that setting the conversational intent on the service or reference element is  
2513 useful when reusing an existing interface definition that contains no SCA information, since it  
2514 requires no modification of the interface artifact.

2515

2516 The meaning of the conversational intent is that both the client and the provider of the interface  
2517 may assume that messages (in either direction) will be handled as part of an ongoing conversation  
2518 without depending on identifying information in the body of the message (i.e. in parameters of the  
2519 operations). In effect, the conversation interface specifies a high-level abstract protocol that must  
2520 be satisfied by any actual binding/policy combination used by the service.

2521 Examples of binding/policy combinations that support conversational interfaces are:

2522 - Web service binding with a WS-RM policy

2523 - Web service binding with a WS-Addressing policy

2524 - Web service binding with a WS-Context policy

2525 - JMS binding with a conversation policy that uses the JMS correlationID header

2526

2527 Conversations occur between one client and one target service. Consequently, requests originating  
2528 from one client to multiple target conversational services will result in multiple conversations. For  
2529 example, if a client A calls services B and C, both of which implement conversational interfaces,  
2530 two conversations result, one between A and B and another between A and C. Likewise, requests  
2531 flowing through multiple implementation instances will result in multiple conversations. For  
2532 example, a request flowing from A to B and then from B to C will involve two conversations (A and  
2533 B, B and C). In the previous example, if a request was then made from C to A, a third  
2534 conversation would result (and the implementation instance for A would be different from the one  
2535 making the original request).

2536 Invocation of any operation of a conversational interface MAY start a conversation. The decision on  
2537 whether an operation would start a conversation depends on the component's implementation and  
2538 its implementation type. Implementation types MAY support components with conversational  
2539 services. If an implementation type does provide this support, it must provide a mechanism for  
2540 determining when a new conversation should be used for an operation (for example, in Java, the  
2541 conversation is new on the first use of an injected reference; in BPEL, the conversation is new  
2542 when the client's partnerLink comes into scope).

2543

2544 One or more operations in a conversational interface may be annotated with an *endsConversation*  
2545 annotation (the mechanism for annotating the interface depends on the interface type). Where an  
2546 interface is **bidirectional**, operations may also be annotated in this way on operations of a  
2547 callback interface. When a conversation ending operation is called, it indicates to both the client  
2548 and the service provider that the conversation is complete. Any subsequent attempts to call an  
2549 operation or a callback operation associated with the same conversation will generate a  
2550 `sca:ConversationViolation` fault.

2551 A `sca:ConversationViolation` fault is thrown when one of the following errors occur:

- 2552 - A message is received for a particular conversation, after the conversation has ended
- 2553 - The conversation identification is invalid (not unique, out of range, etc.)
- 2554 - The conversation identification is not present in the input message of the operation that  
2555 ends the conversation
- 2556 - The client or the service attempts to send a message in a conversation, after the  
2557 conversation has ended

2558 This fault is named within the SCA namespace standard prefix "sca", which corresponds to URI  
2559 <http://docs.oasis-open.org/ns/opencsa/sca/200712>.

2560 The lifecycle of resources and the association between unique identifiers and conversations are  
2561 determined by the service's implementation type and may not be directly affected by the  
2562 "endConversation" annotation. For example, a **WS-BPEL** process **may** outlive most of the  
2563 conversations that it is involved in.

2564 Although conversational interfaces do not require that any identifying information be passed as  
2565 part of the body of messages, there is conceptually an identity associated with the conversation.  
2566 Individual implementations types MAY provide an API to access the ID associated with the  
2567 conversation, although no assumptions may be made about the structure of that identifier.  
2568 Implementation types MAY also provide a means to set the conversation ID by either the client or  
2569 the service provider, although the operation may only be supported by some binding/policy  
2570 combinations.

2571

2572 Implementation-type specifications are encouraged to define and provide conversational instance  
2573 lifecycle management for components that implement conversational interfaces. However,  
2574 implementations may also manage the conversational state manually.

2575

## 2576 8.4 SCA-Specific Aspects for WSDL Interfaces

2577 There are a number of aspects that SCA applies to interfaces in general, such as marking them  
2578 **conversational**. These aspects apply to the interfaces themselves, rather than their use in a  
2579 specific place within SCA. There is thus a need to provide appropriate ways of marking the  
2580 interface definitions themselves, which go beyond the basic facilities provided by the interface  
2581 definition language.

2582 For WSDL interfaces, there is an extension mechanism that permits additional information to be  
2583 included within the WSDL document. SCA takes advantage of this extension mechanism. In order  
2584 to use the SCA extension mechanism, the SCA namespace ([http://docs.oasis-](http://docs.oasis-open.org/ns/opencsa/sca/200712)  
2585 [open.org/ns/opencsa/sca/200712](http://docs.oasis-open.org/ns/opencsa/sca/200712)) must be declared within the WSDL document.

2586 First, SCA defines a global attribute in the SCA namespace which provides a mechanism to attach  
2587 policy intents - **@requires**. The definition of this attribute is as follows:

```
2588 <attribute name="requires" type="sca:listOfQNames"/>
```

2589

```
2590 <simpleType name="listOfQNames">
```

```
2591 <list itemType="QName"/>
```

```
2592 </simpleType>
```

2593 The @requires attribute can be applied to WSDL Port Type elements (WSDL 1.1) and to WSDL  
2594 Interface elements (WSDL 2.0). The attribute contains one or more intent names, as defined by  
2595 [the Policy Framework specification \[10\]](#). Any service or reference that uses an interface with  
2596 required intents implicitly adds those intents to its own @requires list.

2597 To specify that a WSDL interface is conversational, the following attribute setting is used on either  
2598 the WSDL Port Type or WSDL Interface:

```
2599 requires="conversational"
```

2600 SCA defines an **endsConversation** attribute that is used to mark specific operations within a  
2601 WSDL interface declaration as ending a conversation. This only has meaning for WSDL interfaces  
2602 which are also marked conversational. The endsConversation attribute is a global attribute in the  
2603 SCA namespace, with the following definition:

```
2604 <attribute name="endsConversation" type="boolean" default="false"/>
```

2605

2606 The following snippet is an example of a WSDL Port Type annotated with the **requires** attribute on  
2607 the portType and the **endsConversation** attribute on one of the operations:

2608

```
2609 <portType name="LoanService" sca:requires="conversational">
```

```
2610 <operation name="apply">
```

```
2611 <input message="tns:ApplicationInput"/>
```

```
2612 <output message="tns:ApplicationOutput"/>
```

```
2613 </operation>
```

```
2614 <operation name="cancel" sca:endsConversation="true">
```

```
2615 </operation>
```

2616

```
2617 </portType>
```

2618

---

## 2619 9 Binding

2620 Bindings are used by services and references. References use bindings to describe the access  
2621 mechanism used to call a service (which can be a service provided by another SCA composite).  
2622 Services use bindings to describe the access mechanism that clients (which can be a client from  
2623 another SCA composite) have to use to call the service.

2624  
2625 SCA supports the use of multiple different types of bindings. Examples include **SCA service, Web**  
2626 **service, stateless session EJB, data base stored procedure, EIS service**. An SCA runtime  
2627 MUST provide support for SCA service and Web service binding types. SCA provides an  
2628 extensibility mechanism by which an SCA runtime can add support for additional binding types.  
2629 For details on how additional binding types are defined, see the section on the Extension Model.

2630  
2631 A binding is defined by a **binding element** which is a child element of a service or of a reference  
2632 element in a composite. The following snippet shows the composite schema with the schema for  
2633 the binding element.

```
2634  
2635 <?xml version="1.0" encoding="ASCII"?>  
2636 <!-- Bindings schema snippet -->  
2637 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
2638           targetNamespace="xs:anyURI"  
2639           name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?  
2640           constrainingType="QName"?  
2641           requires="list of xs:QName"? policySets="list of  
2642 xs:QName"?>  
2643  
2644     ...  
2645  
2646     <service name="xs:NCName" promote="xs:anyURI"  
2647           requires="list of xs:QName"? policySets="list of xs:QName"?>*  
2648       <interface ... />?  
2649       <binding uri="xs:anyURI"? name="xs:NCName"?  
2650           requires="list of xs:QName"? policySets="list of  
2651 xs:QName"? />*  
2652       <callback?>  
2653           <binding uri="xs:anyURI"? name="xs:NCName"?  
2654               requires="list of xs:QName"?  
2655               policySets="list of xs:QName"? />+  
2656       </callback>  
2657     </service>  
2658  
2659     ...  
2660  
2661     <reference name="xs:NCName" target="list of xs:anyURI"?>
```

```

2662         promote="list of xs:anyURI"? wiredByImpl="xs:boolean"?
2663         multiplicity="0..1 or 1..1 or 0..n or 1..n"?
2664         requires="list of xs:QName"? policySets="list of xs:QName"?>*
2665     <interface ... />?
2666     <binding uri="xs:anyURI"? name="xs:NCName"?
2667         requires="list of xs:QName"? policySets="list of
2668 xs:QName"?/>*
2669     <callback?
2670         <binding uri="xs:anyURI"? name="xs:NCName"?
2671             requires="list of xs:QName"?
2672             policySets="list of xs:QName"?/>+
2673     </callback>
2674 </reference>
2675
2676     ...
2677
2678 </composite>
2679

```

2680 The element name of the binding element is architected; it is in itself a qualified name. The first  
2681 qualifier is always named "binding", and the second qualifier names the respective binding-type  
2682 (e.g. binding.composite, binding.ws, binding.ejb, binding.eis).

2683

2684 A binding element has the following attributes:

- 2685
- 2686 • **uri (optional)** - has the following semantic.
    - 2687 ○ The uri attribute can be omitted.
    - 2688 ○ For a binding of a **reference** the URI attribute defines the target URI of the  
2689 reference. This MUST be either the componentName/serviceName for a wire to an  
2690 endpoint within the SCA domain, or the accessible address of some service  
2691 endpoint either inside or outside the SCA domain (where the addressing scheme is  
2692 defined by the type of the binding).
    - 2693 ○ The circumstances under which the uri attribute can be used are defined in  
2694 section "5.3.1 Specifying the Target Service(s) for a Reference."
    - 2695 ○ For a binding of a **service** the URI attribute defines the URI relative to the  
2696 component, which contributes the service to the SCA domain. The default value for  
2697 the URI is the value of the name attribute of the binding.
  - 2698 • **name (optional)** – a name for the binding instance (an NCName). The name attribute  
2699 allows distinction between multiple binding elements on a single service or reference. The  
2700 default value of the name attribute is the service or reference name. When a service or  
2701 reference has multiple bindings, only one can have the default value; all others must have  
2702 a value specified that is unique within the service or reference. The name also permits the  
2703 binding instance to be referenced from elsewhere – particularly useful for some types of  
2704 binding, which can be declared in a definitions document as a template and referenced  
2705 from other binding instances, simplifying the definition of more complex binding instances  
2706 (see [the JMS Binding specification \[11\]](#) for examples of this referencing).
  - 2707 • **requires (optional)** - a list of policy intents. See the [Policy Framework specification \[10\]](#)  
2708 for a description of this attribute.
  - 2709 • **policySets (optional)** – a list of policy sets. See the [Policy Framework specification \[10\]](#)  
2710 for a description of this attribute.

2710 When multiple bindings exist for an service, it means that the service is available by any of the  
2711 specified bindings. The technique that the SCA runtime uses to choose among available bindings  
2712 is left to the implementation and it may include additional (nonstandard) configuration. Whatever  
2713 technique is used SHOULD be documented.

2714 Services and References can always have their bindings overridden at the SCA domain level,  
2715 unless restricted by Intents applied to them.

2716

2717 If a reference has any bindings they must be "resolved". The bindings MUST include a value for  
2718 the @URI or must otherwise specify an endpoint. The reference MUST NOT be wired using SCA  
2719 mechanisms. To specify constraints on the kinds of bindings that are acceptable for use with a  
2720 reference, the user should specify either policy intents or policy sets.

2721  
2722 Users may also specifically wire, not just to a component service, but to a specific binding offered  
2723 by that target service. To do so, a wire target may optionally be specified with a syntax of  
2724 "componentName/serviceName/bindingName".

2725

2726 The following sections describe the SCA and Web service binding type in detail.

2727

## 2728 9.1 Messages containing Data not defined in the Service Interface

2729

2730 It is possible for a message to include information that is not defined in the interface used to  
2731 define the service, for instance information may be contained in SOAP headers or as MIME  
2732 attachments.

2733 Implementation types MAY make this information available to component implementations in their  
2734 execution context. These implementation types must indicate how this information is accessed  
2735 and in what form they are presented.

2736

## 2737 9.2 Form of the URI of a Deployed Binding

2738

### 2739 9.2.1 Constructing Hierarchical URIs

2740 Bindings that use hierarchical URI schemes construct the effective URI with a combination of the  
2741 following pieces:

2742 Base System URI for a scheme / Component URI / Service Binding URI

2743

2744 Each of these components deserves addition definition:

2745 **Base Domain URI for a scheme.** An SCA domain should define a base URI for each hierarchical  
2746 URI scheme on which it intends to provide services.

2747 For example: the HTTP and HTTPS schemes would each have their own base URI defined for the  
2748 domain. An example of a scheme that is not hierarchical, and therefore will have no base URI is  
2749 the "jms:" scheme.

2750 **Component URI.** The component URI above is for a component that is deployed in the SCA  
2751 Domain. The URI of a component defaults to the name of the component, which is used as a  
2752 relative URI. The component may have a specified URI value. The specified URI value may be an  
2753 absolute URI in which case it becomes the Base URI for all the services belonging to the  
2754 component. If the specified URI value is a relative URI, it is used as the Component URI value  
2755 above.

2756 **Service Binding URI.** The Service Binding URI is the relative URI specified in the "uri" attribute  
2757 of a binding element of the service. The default value of the attribute is value of the binding's  
2758 name attribute treated as a relative URI. If multiple bindings for a single service use the same  
2759 scheme (e.g. HTTP), then only one of the bindings may depend on the default value for the uri  
2760 attribute, i.e. only one may use the default binding name. The service binding URI may also be  
2761 absolute, in which case the absolute URI fully specifies the full URI of the service. Some  
2762 deployment environments may not support the use of absolute URIs in service bindings.

2763 Services deployed into the Domain (as opposed to services of components) have a URI that does  
2764 not include a component name, i.e.:

2765 Base Domain URI for a scheme / Service Binding URI

2766 The name of the containing composite does not contribute to the URI of any service.

2767 For example, a service where the Base URI is "http://acme.com", the component is named  
2768 "stocksComponent" and the service binding name is "getQuote", the URI would look like this:

2769 http://acme.com/stocksComponent/getQuote

2770 Allowing a binding's relative URI to be specified that differs from the name of the service allows  
2771 the URI hierarchy of services to be designed independently of the organization of the domain.

2772 It is good practice to design the URI hierarchy to be independent of the domain organization, but  
2773 there may be times when domains are initially created using the default URI hierarchy. When this  
2774 is the case, the organization of the domain can be changed, while maintaining the form of the URI  
2775 hierarchy, by giving appropriate values to the **uri** attribute of select elements. Here is an example  
2776 of a change that can be made to the organization while maintaining the existing URIs:

2777 To move a subset of the services out of one component (say "foo") to a new component (say  
2778 "bar"), the new component should have bindings for the moved services specify a URI  
2779 "../foo/MovedService"..

2780 The URI attribute may also be used in order to create shorter URIs for some endpoints, where the  
2781 component name may not be present in the URI at all. For example, if a binding has a **uri**  
2782 attribute of "../myService" the component name will not be present in the URI.

## 2783 9.2.2 Non-hierarchical URIs

2784 Bindings that use non-hierarchical URI schemes (such as jms: or mailto:) may optionally make  
2785 use of the "uri" attribute, which is the complete representation of the URI for that service  
2786 binding. Where the binding does not use the "uri" attribute, the binding must offer a different  
2787 mechanism for specifying the service address.

## 2788 9.2.3 Determining the URI scheme of a deployed binding

2789 One of the things that needs to be determined when building the effective URI of a deployed  
2790 binding (i.e. endpoint) is the URI scheme. The process of determining the endpoint URI scheme is  
2791 binding type specific.

2792 If the binding type supports a single protocol then there is only one URI scheme associated with it.  
2793 In this case, that URI scheme is used.

2794 If the binding type supports multiple protocols, the binding type implementation determines the  
2795 URI scheme by introspecting the binding configuration, which may include the policy sets  
2796 associated with the binding.

2797 A good example of a binding type that supports multiple protocols is binding.ws, which can be  
2798 configured by referencing either an "abstract" WSDL element (i.e. portType or interface) or a  
2799 "concrete" WSDL element (i.e. binding, port or endpoint). When the binding references a PortType  
2800 or Interface, the protocol and therefore the URI scheme is derived from the intents/policy sets  
2801 attached to the binding. When the binding references a "concrete" WSDL element, there are two  
2802 cases:

- 2803 1) The referenced WSDL binding element uniquely identifies a URI scheme. This is the most  
2804 common case. In this case, the URI scheme is given by the protocol/transport specified in the  
2805 WSDL binding element.
- 2806 2) The referenced WSDL binding element doesn't uniquely identify a URI scheme. For example,  
2807 when HTTP is specified in the @transport attribute of the SOAP binding element, both "http"  
2808 and "https" could be used as valid URI schemes. In this case, the URI scheme is determined  
2809 by looking at the policy sets attached to the binding.

2810 It's worth noting that an intent supported by a binding type may completely change the behavior  
2811 of the binding. For example, when the intent "confidentiality/transport" is required by an HTTP  
2812 binding, SSL is turned on. This basically changes the URI scheme of the binding from "http" to  
2813 "https".

2814

## 2815 9.3 SCA Binding

2816 The SCA binding element is defined by the following schema.

2817

```
2818 <binding.sca />
```

2819

2820 The SCA binding can be used for service interactions between references and services contained  
2821 within the SCA domain. The way in which this binding type is implemented is not defined by the  
2822 SCA specification and it can be implemented in different ways by different SCA runtimes. The only  
2823 requirement is that the required qualities of service must be implemented for the SCA binding  
2824 type. The SCA binding type is **not** intended to be an interoperable binding type. For  
2825 interoperability, an interoperable binding type such as the Web service binding should be used.

2826 A service definition with no binding element specified uses the SCA binding.  
2827 <binding.sca/> would only have to be specified in override cases, or when you specify a  
2828 set of bindings on a service definition and the SCA binding should be one of them.

2829 If a reference does not have a binding, then the binding used can be any of the bindings  
2830 specified by the service provider, as long as the intents required by the reference and  
2831 the service are all respected.

2832 If the interface of the service or reference is local, then the local variant of the SCA  
2833 binding will be used. If the interface of the service or reference is remotable, then either  
2834 the local or remote variant of the SCA binding will be used depending on whether source  
2835 and target are co-located or not.

2836 If a reference specifies an URI via its uri attribute, then this provides the default wire to a service  
2837 provided by another domain level component. The value of the URI has to be as follows:

- 2838 • <domain-component-name>/<service-name>

2839

### 2840 9.3.1 Example SCA Binding

2841 The following snippet shows the MyValueComposite.composite file for the MyValueComposite  
2842 containing the service element for the MyValueService and a reference element for the  
2843 StockQuoteService. Both the service and the reference use an SCA binding. The target for the  
2844 reference is left undefined in this binding and would have to be supplied by the composite in which  
2845 this composite is used.

2846

```
2847 <?xml version="1.0" encoding="ASCII"?>
```

```
2848 <!-- Binding SCA example -->
```

```
2849     <composite      xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2850                   targetNamespace="http://foo.com"
2851                   name="MyValueComposite" >
2852
2853         <service name="MyValueService" promote="MyValueComponent" >
2854             <interface.java interface="services.myvalue.MyValueService" />
2855             <binding.sca/>
2856             ...
2857         </service>
2858
2859         ...
2860
2861         <reference name="StockQuoteService"
2862                 promote="MyValueComponent/StockQuoteReference" >
2863             <interface.java
2864                 interface="services.stockquote.StockQuoteService" />
2865             <binding.sca/>
2866         </reference>
2867
2868     </composite>
2869
```

## 2870 9.4 Web Service Binding

2871 SCA defines a Web services binding. This is described in [a separate specification document \[9\]](#).

2872

## 2873 9.5 JMS Binding

2874 SCA defines a JMS binding. This is described in [a separate specification document \[11\]](#).

2875

## 10 SCA Definitions

2876 There are a variety of SCA artifacts which are generally useful and which are not specific to a  
2877 particular composite or a particular component. These shared artifacts include intents, policy sets,  
2878 bindings, binding type definitions and implementation type definitions.

2879 All of these artifacts within an SCA Domain are defined in a global, SCA Domain-wide file named  
2880 definitions.xml. The definitions.xml file contains a definitions element that conforms to the  
2881 following pseudo-schema snippet:

```
2882 <?xml version="1.0" encoding="ASCII"?>
2883 <!-- Composite schema snippet -->
2884 <definitions xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2885             targetNamespace="xs:anyURI">
2886
2887     <sca:intent/*
2888
2889     <sca:policySet/*
2890
2891     <sca:binding/*
2892
2893     <sca:bindingType/*
2894
2895     <sca:implementationType/*
2896
2897 </definitions>
```

2898 The definitions element has the following attribute:

- 2899 • **targetNamespace (required)** – the namespace into which the child elements of this  
2900 definitions element are placed (used for artifact resolution)

2901 The definitions element contains optional child elements – intent, policySet, binding, bindingtype  
2902 and implementationType. These elements are described elsewhere in this specification or in [the](#)  
2903 [SCA Policy Framework specification \[10\]](#). The use of the elements declared within a definitions  
2904 element is described in the SCA Policy Framework specification [10] and in [the JMS Binding](#)  
2905 [specification \[11\]](#).

2906

2907

## 11 Extension Model

2908

2909 The assembly model can be extended with support for new interface types, implementation types  
2910 and binding types. The extension model is based on XML schema substitution groups. There are  
2911 three XML Schema substitution group heads defined in the SCA namespace: **interface**,  
2912 **implementation** and **binding**, for interface types, implementation types and binding types,  
2913 respectively.

2914 The SCA Client and Implementation specifications and the SCA Bindings specifications (see [1],  
2915 [9], [11]) use these XML Schema substitution groups to define some basic types of interfaces,  
2916 implementations and bindings, but other types can be defined as required, where support for  
2917 these extra ones is available from the runtime. The interface type elements, implementation type  
2918 elements, and binding type elements defined by the SCA specifications are all part of the SCA  
2919 namespace ("http://docs.oasis-open.org/ns/opencsa/sca/200712"), as indicated in their  
2920 respective schemas. New interface types, implementation types and binding types that are defined  
2921 using this extensibility model, which are not part of these SCA specifications must be defined in  
2922 namespaces other than the SCA namespace.

2923 The "." notation is used in naming elements defined by the SCA specifications ( e.g.  
2924 <implementation.java ... />, <interface.wsdl ... />, <binding.ws ... />), not as a parallel  
2925 extensibility approach but as a naming convention that improves usability of the SCA assembly  
2926 language.

2927

2928 **Note:** How to contribute SCA model extensions and their runtime function to an SCA runtime will  
2929 be defined by a future version of the specification.

2930

### 11.1 Defining an Interface Type

2932 The following snippet shows the base definition for the **interface** element and **Interface** type  
2933 contained in **sca-core.xsd**; see appendix for complete schema.

2934

```
2935 <?xml version="1.0" encoding="UTF-8"?>  
2936 <!-- (c) Copyright SCA Collaboration 2006 -->  
2937 <schema xmlns="http://www.w3.org/2001/XMLSchema"  
2938         targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
2939         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
2940         elementFormDefault="qualified">
```

2941

2942 ...

2943

```
2944     <element name="interface" type="sca:Interface" abstract="true"/>  
2945     <complexType name="Interface"/>  
2946     <complexType name="Interface" abstract="true">  
2947         <attribute name="requires" type="sca:listOfQNames" use="optional"/>  
2948         <attribute name="policySets" type="sca:listOfQNames" use="optional"/>  
2949     </complexType>
```

2950

2951  
2952       ...  
2953  
2954       </schema>

2955       In the following snippet we show how the base definition is extended to support Java interfaces.  
2956       The snippet shows the definition of the **interface.java** element and the **JavaInterface** type  
2957       contained in **sca-interface-java.xsd**.

2958  
2959       <?xml version="1.0" encoding="UTF-8"?>  
2960       <schema xmlns="http://www.w3.org/2001/XMLSchema"  
2961               targetNamespace="http://docs.oasis-  
2962       open.org/ns/opencsa/sca/200712"  
2963               xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712">  
2964  
2965           <element name="interface.java" type="sca:JavaInterface"  
2966               substitutionGroup="sca:interface"/>  
2967           <complexType name="JavaInterface">  
2968               <complexContent>  
2969                   <extension base="sca:Interface">  
2970                       <attribute name="interface" type="NCName"  
2971       use="required"/>  
2972                   </extension>  
2973               </complexContent>  
2974           </complexType>  
2975       </schema>

2976       In the following snippet we show an example of how the base definition can be extended by other  
2977       specifications to support a new interface not defined in the SCA specifications. The snippet shows  
2978       the definition of the **my-interface-extension** element and the **my-interface-extension-type**  
2979       type.

2980       <?xml version="1.0" encoding="UTF-8"?>  
2981       <schema xmlns="http://www.w3.org/2001/XMLSchema"  
2982               targetNamespace="http://www.example.org/myextension"  
2983               xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
2984               xmlns:tns="http://www.example.org/myextension">  
2985  
2986           <element name="my-interface-extension" type="tns:my-interface-  
2987       extension-type"  
2988               substitutionGroup="sca:interface"/>  
2989           <complexType name="my-interface-extension-type">  
2990               <complexContent>  
2991                   <extension base="sca:Interface">  
2992                       ...  
2993                   </extension>  
2994               </complexContent>  
2995           </complexType>

2996 </schema>  
2997

## 2998 11.2 Defining an Implementation Type

2999 The following snippet shows the base definition for the *implementation* element and  
3000 **Implementation** type contained in *sca-core.xsd*; see appendix for complete schema.

```
3001  
3002 <?xml version="1.0" encoding="UTF-8"?>  
3003 <!-- (c) Copyright SCA Collaboration 2006 -->  
3004 <schema xmlns="http://www.w3.org/2001/XMLSchema"  
3005         targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
3006         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
3007         elementFormDefault="qualified">  
3008  
3009     ...  
3010  
3011     <element name="implementation" type="sca:Implementation"  
3012 abstract="true"/>  
3013     <complexType name="Implementation"/>  
3014  
3015     ...  
3016  
3017 </schema>
```

3018  
3019 In the following snippet we show how the base definition is extended to support Java  
3020 implementation. The snippet shows the definition of the *implementation.java* element and the  
3021 **JavaImplementation** type contained in *sca-implementation-java.xsd*.

```
3022  
3023 <?xml version="1.0" encoding="UTF-8"?>  
3024 <schema xmlns="http://www.w3.org/2001/XMLSchema"  
3025         targetNamespace="http://docs.oasis-  
3026 open.org/ns/opencsa/sca/200712"  
3027         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712">  
3028  
3029     <element name="implementation.java" type="sca:JavaImplementation"  
3030             substitutionGroup="sca:implementation"/>  
3031     <complexType name="JavaImplementation">  
3032         <complexContent>  
3033             <extension base="sca:Implementation">  
3034                 <attribute name="class" type="NCName"  
3035 use="required"/>  
3036             </extension>  
3037         </complexContent>  
3038     </complexType>  
3039 </schema>
```

3040 In the following snippet we show an example of how the base definition can be extended by other  
3041 specifications to support a new implementation type not defined in the SCA specifications. The  
3042 snippet shows the definition of the **my-impl-extension** element and the **my-impl-extension-**  
3043 **type** type.

```
3044 <?xml version="1.0" encoding="UTF-8"?>
3045 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3046         targetNamespace="http://www.example.org/myextension"
3047         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3048         xmlns:tns="http://www.example.org/myextension">
3049
3050     <element name="my-impl-extension" type="tns:my-impl-extension-type"
3051             substitutionGroup="sca:implementation"/>
3052     <complexType name="my-impl-extension-type">
3053         <complexContent>
3054             <extension base="sca:Implementation">
3055                 ...
3056             </extension>
3057         </complexContent>
3058     </complexType>
3059 </schema>
```

3061 In addition to the definition for the new implementation instance element, there needs to be an  
3062 associated implementationType element which provides metadata about the new implementation  
3063 type. The pseudo schema for the implementationType element is shown in the following snippet:

```
3064 <implementationType type="xs:QName"
3065                    alwaysProvides="list of intent xs:QName"
3066                    mayProvide="list of intent xs:QName"/>
```

3068 The implementation type has the following attributes:

- 3069 • **type (required)** – the type of the implementation to which this implementationType  
3070 element applies. This is intended to be the QName of the implementation element for the  
3071 implementation type, such as "sca:implementation.java"
- 3072 • **alwaysProvides (optional)** – a set of intents which the implementation type always  
3073 provides. See [the Policy Framework specification \[10\]](#) for details.
- 3074 • **mayProvide (optional)** – a set of intents which the implementation type may provide.  
3075 See [the Policy Framework specification \[10\]](#) for details.

3076

## 3077 11.3 Defining a Binding Type

3078 The following snippet shows the base definition for the **binding** element and **Binding** type  
3079 contained in **sca-core.xsd**; see appendix for complete schema.

3080

```
3081 <?xml version="1.0" encoding="UTF-8"?>
3082 <!-- binding type schema snippet -->
3083 <!-- (c) Copyright SCA Collaboration 2006, 2007 -->
```

```

3084 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3085         targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3086         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3087         elementFormDefault="qualified">
3088
3089     ...
3090
3091     <element name="binding" type="sca:Binding" abstract="true"/>
3092     <complexType name="Binding">
3093         <attribute name="uri" type="anyURI" use="optional"/>
3094         <attribute name="name" type="NCName" use="optional"/>
3095         <attribute name="requires" type="sca:listOfQNames"
3096 use="optional"/>
3097         <attribute name="policySets" type="sca:listOfQNames"
3098 use="optional"/>
3099     </complexType>
3100
3101     ...
3102
3103 </schema>

```

3104 In the following snippet we show how the base definition is extended to support Web service  
3105 binding. The snippet shows the definition of the **binding.ws** element and the  
3106 **WebServiceBinding** type contained in **sca-binding-webservice.xsd**.

```

3107
3108 <?xml version="1.0" encoding="UTF-8"?>
3109 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3110         targetNamespace="http://docs.oasis-
3111 open.org/ns/opencsa/sca/200712"
3112         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712">
3113
3114     <element name="binding.ws" type="sca:WebServiceBinding"
3115         substitutionGroup="sca:binding"/>
3116     <complexType name="WebServiceBinding">
3117         <complexContent>
3118             <extension base="sca:Binding">
3119                 <attribute name="port" type="anyURI" use="required"/>
3120             </extension>
3121         </complexContent>
3122     </complexType>
3123 </schema>

```

3124 In the following snippet we show an example of how the base definition can be extended by other  
3125 specifications to support a new binding not defined in the SCA specifications. The snippet shows  
3126 the definition of the **my-binding-extension** element and the **my-binding-extension-type** type.

```

3127 <?xml version="1.0" encoding="UTF-8"?>
3128 <schema xmlns="http://www.w3.org/2001/XMLSchema"

```

```

3129         targetNamespace="http://www.example.org/myextension"
3130         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3131         xmlns:tns="http://www.example.org/myextension" >
3132
3133         <element name="my-binding-extension" type="tns:my-binding-extension-
3134         type"
3135             substitutionGroup="sca:binding"/>
3136         <complexType name="my-binding-extension-type">
3137             <complexContent>
3138                 <extension base="sca:Binding">
3139                     ...
3140                 </extension>
3141             </complexContent>
3142         </complexType>
3143 </schema>
3144

```

3145 In addition to the definition for the new binding instance element, there needs to be an associated  
3146 bindingType element which provides metadata about the new binding type. The pseudo schema  
3147 for the bindingType element is shown in the following snippet:

```

3148 <bindingType type="xs:QName"
3149             alwaysProvides="list of intent QNames"?
3150             mayProvide = "list of intent QNames"?/>
3151

```

3152 The binding type has the following attributes:

- 3153 • **type (required)** – the type of the binding to which this bindingType element applies.  
3154 This is intended to be the QName of the binding element for the binding type, such as  
3155 "sca:binding.ws"
- 3156 • **alwaysProvides (optional)** – a set of intents which the binding type always provides.  
3157 See [the Policy Framework specification \[10\]](#) for details.
- 3158 • **mayProvide (optional)** – a set of intents which the binding type may provide. See [the](#)  
3159 [Policy Framework specification \[10\]](#) for details.

---

## 3160 12 Packaging and Deployment

### 3161 12.1 Domains

3162 An **SCA Domain** represents a complete runtime configuration, potentially distributed over a series  
3163 of interconnected runtime nodes.

3164 A single SCA domain defines the boundary of visibility for all SCA mechanisms. For example, SCA  
3165 wires can only be used to connect components within a single SCA domain. Connections to  
3166 services outside the domain must use binding specific mechanisms for addressing services (such  
3167 as WSDL endpoint URIs). Also, SCA mechanisms such as intents and policySets can only be used  
3168 in the context of a single domain. In general, external clients of a service that is developed and  
3169 deployed using SCA should not be able to tell that SCA was used to implement the service – it is  
3170 an implementation detail.

3171 The size and configuration of an SCA Domain is not constrained by the SCA Assembly specification  
3172 and is expected to be highly variable. An SCA Domain typically represents an area of business  
3173 functionality controlled by a single organization. For example, an SCA Domain may be the whole  
3174 of a business, or it may be a department within a business.

3175 As an example, for the accounts department in a business, the SCA Domain might cover all  
3176 finance-related functions, and it might contain a series of composites dealing with specific areas of  
3177 accounting, with one for Customer accounts and another dealing with Accounts Payable.

3178 An SCA domain has the following:

- 3179 • A virtual domain-level composite whose components are deployed and running
- 3180 • A set of *installed contributions* that contain implementations, interfaces and other artifacts  
3181 necessary to execute components
- 3182 • A set of logical services for manipulating the set of contributions and the virtual domain-  
3183 level composite.

3184 The information associated with an SCA domain can be stored in many ways, including but not  
3185 limited to a specific filesystem structure or a repository.

### 3186 12.2 Contributions

3187 An SCA domain may require a large number of different artifacts in order to work. These artifacts  
3188 include artifacts defined by SCA and other artifacts such as object code files and interface  
3189 definition files. The SCA-defined artifact types are all XML documents. The root elements of the  
3190 different SCA definition documents are: *composite*, *componentType*, *constrainingType* and  
3191 *definitions*. XML artifacts that are not defined by SCA but which may be needed by an SCA  
3192 domain include XML Schema documents, WSDL documents, and BPEL documents. SCA  
3193 constructs, like other XML-defined constructs, use XML qualified names for their identity (i.e.  
3194 namespace + local name).

3195 Non-XML artifacts are also required within an SCA domain. The most obvious examples of such  
3196 non-XML artifacts are Java, C++ and other programming language files necessary for component  
3197 implementations. Since SCA is extensible, other XML and non-XML artifacts may also be required.

3198 SCA defines an interoperable packaging format for contributions (ZIP), as specified below. This  
3199 format is not the only packaging format that an SCA runtime can use. SCA allows many different  
3200 packaging formats, but requires that the ZIP format be supported. When using the ZIP format for  
3201 deploying a contribution, this specification does not specify whether that format is retained after  
3202 deployment. For example, a Java EE based SCA runtime may convert the ZIP package to an EAR  
3203 package. SCA expects certain characteristics of any packaging:

- 3204 • It must be possible to present the artifacts of the packaging to SCA as a hierarchy of  
3205 resources based off of a single root

- 3206
- A directory resource should exist at the root of the hierarchy named META-INF
- 3207
- A document should exist directly under the META-INF directory named sca-
- 3208
- contribution.xml which lists the SCA Composites within the contribution that are runnable.
- 3209

3210 The same document also optionally lists namespaces of constructs that are defined within  
3211 the contribution and which may be used by other contributions  
3212 Optionally, additional elements may exist that list the namespaces of constructs that are  
3213 needed by the contribution and which must be found elsewhere, for example in other  
3214 contributions. These optional elements may not be physically present in the packaging,  
3215 but may be generated based on the definitions and references that are present, or they  
3216 may not exist at all if there are no unresolved references.

3217

3218 See the section "SCA Contribution Metadata Document" for details of the format of this  
3219 file.

3220 To illustrate that a variety of packaging formats can be used with SCA, the following are examples  
3221 of formats that might be used to package SCA artifacts and metadata (as well as other artifacts)  
3222 as a contribution:

- 3223
- A filesystem directory
- 3224
- An OSGi bundle
- 3225
- A compressed directory (zip, gzip, etc)
- 3226
- A JAR file (or its variants – WAR, EAR, etc)

3227 Contributions do not contain other contributions. If the packaging format is a JAR file that  
3228 contains other JAR files (or any similar nesting of other technologies), the internal files are not  
3229 treated as separate SCA contributions. It is up to the implementation to determine whether the  
3230 internal JAR file should be represented as a single artifact in the contribution hierarchy or whether  
3231 all of the contents should be represented as separate artifacts.

3232 A goal of SCA's approach to deployment is that the contents of a contribution should not need to  
3233 be modified in order to install and use the contents of the contribution in a domain.

3234

## 3235 12.2.1 SCA Artifact Resolution

3236 Contributions may be self-contained, in that all of the artifacts necessary to run the contents of  
3237 the contribution are found within the contribution itself. However, it may also be the case that the  
3238 contents of the contribution make one or many references to artifacts that are not contained  
3239 within the contribution. These references may be to SCA artifacts or they may be to other  
3240 artifacts such as WSDL files, XSD files or to code artifacts such as Java class files and BPEL scripts.

3241 A contribution may use some artifact-related or packaging-related means to resolve artifact  
3242 references. Examples of such mechanisms include:

- 3243
- wsdlLocation and schemaLocation attributes in references to WSDL and XSD schema  
3244 artifacts respectively
- 3245
- OSGi bundle mechanisms for resolving Java class and related resource dependencies

3246 Where present, these mechanisms must be used to resolve artifact dependencies.

3247 SCA also provides an artifact resolution mechanism. The SCA artifact resolution mechanisms are  
3248 used either where no other mechanisms are available, or in cases where the mechanisms used by  
3249 the various contributions in the same SCA Domain are different. An example of the latter case is  
3250 where an OSGi Bundle is used for one contribution but where a second contribution used by the  
3251 first one is not implemented using OSGi - eg the second contribution is a mainframe COBOL  
3252 service whose interfaces are declared using WSDL which must be accessed by the first  
3253 contribution.

3254 The SCA artifact resolution is likely to be most useful for SCA domains containing heterogeneous  
3255 mixtures of contribution, where artifact-related or packaging-related mechanisms are unlikely to  
3256 work across different kinds of contribution.

3257 SCA artifact resolution works on the principle that a contribution which needs to use artifacts  
3258 defined elsewhere expresses these dependencies using **import** statements in metadata belonging  
3259 to the contribution. A contribution controls which artifacts it makes available to other  
3260 contributions through **export** statements in metadata attached to the contribution.

3261

## 3262 12.2.2 SCA Contribution Metadata Document

3263 The contribution optionally contains a document that declares runnable composites, exported  
3264 definitions and imported definitions. The document is found at the path of META-INF/sca-  
3265 contribution.xml relative to the root of the contribution. Frequently some SCA metadata may  
3266 need to be specified by hand while other metadata is generated by tools (such as the <import>  
3267 elements described below). To accommodate this, it is also possible to have an identically  
3268 structured document at META-INF/sca-contribution-generated.xml. If this document exists (or is  
3269 generated on an as-needed basis), it will be merged into the contents of sca-contribution.xml,  
3270 with the entries in sca-contribution.xml taking priority if there are any conflicting declarations.

3271

3272 The format of the document is:

```
3273 <?xml version="1.0" encoding="ASCII"?>  
3274 <!-- sca-contribution pseudo-schema -->  
3275 <contribution xmlns=http://docs.oasis-open.org/ns/opencsa/sca/200712>  
3276  
3277     <deployable composite="xs:QName"/>*  
3278     <import namespace="xs:String" location="xs:AnyURI"?/>*  
3279     <export namespace="xs:String"/>*  
3280  
3281 </contribution>
```

3282

3283 **deployable element:** Identifies a composite which is a composite within the contribution that is a  
3284 composite intended for potential inclusion into the virtual domain-level composite. Other  
3285 composites in the contribution are not intended for inclusion but only for use by other composites.  
3286 New composites can be created for a contribution after it is installed, by using the [add Deployment  
3287 Composite](#) capability and the add To Domain Level Composite capability.

3288 

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

3289

3290 **Export element:** A declaration that artifacts belonging to a particular namespace are exported  
3291 and are available for use within other contributions. An export declaration in a contribution  
3292 specifies a namespace, all of whose definitions are considered to be exported. By default,  
3293 definitions are not exported.

3294 The SCA artifact export is useful for SCA domains containing heterogeneous mixtures of  
3295 contribution packagings and technologies, where artifact-related or packaging-related mechanisms  
3296 are unlikely to work across different kinds of contribution.

3297 

- **namespace (required)** – For XML definitions, which are identified by QNames, the  
3298 namespace should be the namespace URI for the exported definitions. For XML  
3299 technologies that define multiple *symbol spaces* that can be used within one namespace  
3300 (e.g. WSDL port types are a different symbol space from WSDL bindings), all definitions  
3301 from all symbol spaces are exported.

3302

3303 Technologies that use naming schemes other than QNames must use a different export

3304 element from the same substitution group as the the SCA <export> element. The  
3305 element used identifies the technology, and may use any value for the namespace that is  
3306 appropriate for that technology. For example, <export.java> can be used can be used to  
3307 export java definitions, in which case the namespace should be a fully qualified package  
3308 name.

3309  
3310 **Import element:** Import declarations specify namespaces of definitions that are needed by the  
3311 definitions and implementations within the contribution, but which are not present in the  
3312 contribution. It is expected that in most cases import declarations will be generated based on  
3313 introspection of the contents of the contribution. In this case, the import declarations would be  
3314 found in the META-INF/ sca-contribution-generated.xml document.

3315

- **namespace (required)** – For XML definitions, which are identified by QNames, the  
3316 namespace should be the namespace URI for the imported definitions. For XML  
3317 technologies that define multiple *symbol spaces* that can be used within one namespace  
3318 (e.g. WSDL port types are a different symbol space from WSDL bindings), all definitions  
3319 from all symbol spaces are imported.

3320  
3321 Technologies that use naming schemes other than QNames must use a different import  
3322 element from the same substitution group as the the SCA <import> element. The  
3323 element used identifies the technology, and may use any value for the namespace that is  
3324 appropriate for that technology. For example, <import.java> can be used can be used to  
3325 import java definitions, in which case the namespace should be a fully qualified package  
3326 name.

3327

- **location (optional)** – a URI to resolve the definitions for this import. SCA makes no  
3328 specific requirements for the form of this URI, nor the means by which it is resolved. It  
3329 may point to another contribution (through its URI) or it may point to some location  
3330 entirely outside the SCA Domain.

3331

3332 It is expected that SCA runtimes may define implementation specific ways of resolving location  
3333 information for artifact resolution between contributions. These mechanisms will however usually  
3334 be limited to sets of contributions of one runtime technology and one hosting environment.

3335 In order to accommodate imports of artifacts between contributions of disparate runtime  
3336 technologies, it is strongly suggested that SCA runtimes honor SCA contribution URIs as location  
3337 specification.

3338 SCA runtimes that support contribution URIs for cross-contribution resolution of SCA artifacts  
3339 should do so similarly when used as @schemaLocation and @wsdlLocation and other artifact  
3340 location specifications.

3341 The order in which the import statements are specified may play a role in this mechanism. Since  
3342 definitions of one namespace can be distributed across several artifacts, multiple import  
3343 declarations can be made for one namespace.

3344

3345 The location value is only a default, and dependent contributions listed in the call to  
3346 installContribution should override the value if there is a conflict. However, the specific  
3347 mechanism for resolving conflicts between contributions that define conflicting definitions is  
3348 implementation specific.

3349

3350 If the value of the location attribute is an SCA contribution URI, then the contribution packaging  
3351 may become dependent on the deployment environment. In order to avoid such a dependency,  
3352 dependent contributions should be specified only when deploying or updating contributions as  
3353 specified in the section 'Operations for Contributions' below.

### 3354 12.2.3 Contribution Packaging using ZIP

3355 SCA allows many different packaging formats that SCA runtimes can support, but SCA requires  
3356 that all runtimes support the ZIP packaging format for contributions. This format allows that

3357 metadata specified by the section 'SCA Contribution Metadata Document' be present. Specifically,  
3358 it may contain a top-level "META-INF" directory and a "META-INF/sca-contribution.xml" file and  
3359 there may also be an optional "META-INF/sca-contribution-generated.xml" file in the package. SCA  
3360 defined artifacts as well as non-SCA defined artifacts such as object files, WSDL definition, Java  
3361 classes may be present anywhere in the ZIP archive,

3362 A up to date definition of the ZIP file format is published by PKWARE in [an Application Note on the](#)  
3363 [.ZIP file format \[12\]](#).

3364

## 3365 12.3 Installed Contribution

3366 As noted in the section above, the contents of a contribution should not need to be modified in  
3367 order to install and use it within a domain. An *installed contribution* is a contribution with all of  
3368 the associated information necessary in order to execute *deployable composites* within the  
3369 contribution.

3370 An installed contribution is made up of the following things:

- 3371 • Contribution Packaging – the contribution that will be used as the starting point for  
3372 resolving all references
- 3373 • Contribution base URI
- 3374 • Dependent contributions: a set of snapshots of other contributions that are used to resolve  
3375 the import statements from the root composite and from other dependent contributions
  - 3376 ○ Dependent contributions may or may not be shared with other installed  
3377 contributions.
  - 3378 ○ When the snapshot of any contribution is taken is implementation defined, ranging  
3379 from the time the contribution is installed to the time of execution
- 3380 • Deployment-time composites.  
3381 These are composites that are added into an installed contribution after it has been  
3382 deployed. This makes it possible to provide final configuration and access to  
3383 implementations within a contribution without having to modify the contribution. These  
3384 are optional, as composites that already exist within the contribution may also be used for  
3385 deployment.

3386

3387 Installed contributions provide a context in which to resolve qualified names (e.g. QNames in XML,  
3388 fully qualified class names in Java).

3389 If multiple dependent contributions have exported definitions with conflicting qualified names, the  
3390 algorithm used to determine the qualified name to use is implementation dependent.  
3391 Implementations of SCA may also generate an error if there are conflicting names.

3392

### 3393 12.3.1 Installed Artifact URIs

3394 When a contribution is installed, all artifacts within the contribution are assigned URIs, which are  
3395 constructed by starting with the base URI of the contribution and adding the relative URI of each  
3396 artifact (recalling that SCA requires that any packaging format be able to offer up its artifacts in a  
3397 single hierarchy).

3398

## 3399 12.4 Operations for Contributions

3400 SCA Domains provide the following conceptual functionality associated with contributions  
3401 (meaning the function may not be represented as addressable services and also meaning that

3402 equivalent functionality may be provided in other ways). The functionality is optional meaning that  
3403 some SCA runtimes may choose not to provide that functionality in any way:

### 3404 12.4.1 install Contribution & update Contribution

3405  
3406 Creates or updates an installed contribution with a supplied root contribution, and installed at a  
3407 supplied base URI. A supplied dependent contribution list specifies the contributions that should  
3408 be used to resolve the dependencies of the root contribution and other dependent contributions.  
3409 These override any dependent contributions explicitly listed via the location attribute in the import  
3410 statements of the contribution.

3411  
3412 SCA follows the simplifying assumption that the use of a contribution for resolving anything also  
3413 means that all other exported artifacts can be used from that contribution. Because of this, the  
3414 dependent contribution list is just a list of installed contribution URIs. There is no need to specify  
3415 what is being used from each one.

3416 Each dependent contribution is also an installed contribution, with its own dependent  
3417 contributions. By default these dependent contributions of the dependent contributions (which we  
3418 will call *indirect dependent contributions*) are included as dependent contributions of the installed  
3419 contribution. However, if a contribution in the dependent contribution list exports any conflicting  
3420 definitions with an indirect dependent contribution, then the indirect dependent contribution is not  
3421 included (i.e. the explicit list overrides the default inclusion of indirect dependent contributions).  
3422 Also, if there is ever a conflict between two indirect dependent contributions, then the conflict  
3423 must be resolved by an explicit entry in the dependent contribution list.

3424 Note that in many cases, the dependent contribution list can be generated. In particular, if a  
3425 domain is careful to avoid creating duplicate definitions for the same qualified name, then it is  
3426 easy for this list to be generated by tooling.

### 3427 12.4.2 add Deployment Composite & update Deployment Composite

3428 Adds or updates a deployment composite using a supplied composite ("composite by value" – a  
3429 data structure, not an existing resource in the domain) to the contribution identified by a supplied  
3430 contribution URI. The added or updated deployment composite is given a relative URI that  
3431 matches the @name attribute of the composite, with a ".composite" suffix. Since all composites  
3432 must run within the context of a installed contribution (any component implementations or other  
3433 definitions are resolved within that contribution), this functionality makes it possible for the  
3434 deployer to create a composite with final configuration and wiring decisions and add it to an  
3435 installed contribution without having to modify the contents of the root contribution.

3436 Also, in some use cases, a contribution may include only implementation code (e.g. PHP scripts).  
3437 It should then be possible for those to be given component names by a (possibly generated)  
3438 composite that is added into the installed contribution, without having to modify the packaging.

### 3439 12.4.3 remove Contribution

3440 Removes the deployed contribution identified by a supplied contribution URI.

3441

## 3442 12.5 Use of Existing (non-SCA) Mechanisms for Resolving Artifacts

3443

3444 For certain types of artifact, there are existing and commonly used mechanisms for referencing a  
3445 specific concrete location where the artifact can be resolved.

3446 Examples of these mechanisms include:

- 3447 • For WSDL files, the **@wsdlLocation** attribute is a hint that has a URI value pointing to the  
3448 place holding the WSDL itself.

- 3449           • For XSDs, the **@schemaLocation** attribute is a hint which matches the namespace to a  
3450           URI where the XSD is found.

3451       **Note:** In neither of these cases is the runtime obliged to use the location hint and the URI does  
3452       not have to be dereferenced.

3453       SCA permits the use of these mechanisms. Where present, these mechanisms take precedence  
3454       over the SCA mechanisms. However, use of these mechanisms is discouraged because tying  
3455       assemblies to addresses in this way makes the assemblies less flexible and prone to errors when  
3456       changes are made to the overall SCA Domain.

3457       **Note:** If one of these mechanisms is present, but there is a failure to find the resource indicated  
3458       when using the mechanism (eg the URI is incorrect or invalid, say) the SCA runtime **MUST** raise  
3459       an error and **MUST NOT** attempt to use SCA resolution mechanisms as an alternative.

3460

## 3461   12.6 Domain-Level Composite

3462       The domain-level composite is a virtual composite, in that it is not defined by a composite  
3463       definition document. Rather, it is built up and modified through operations on the domain.  
3464       However, in other respects it is very much like a composite, since it contains components, wires,  
3465       services and references.

3466

3467       The value of @autowire for the logical domain composite **MUST** be autowire="false".

3468

3469       For components at the Domain level, with References for which @autowire="true" applies, the  
3470       behaviour of the SCA runtime for a given Domain **MUST** take ONE of the 3 following forms:

- 3471           1. The SCA runtime **MAY** disallow deployment of any components with autowire References.  
3472           In this case, the SCA runtime **MUST** generate an exception at the point where the  
3473           component is deployed.
- 3474           2. The SCA runtime **MAY** evaluate the target(s) for the reference at the time that the  
3475           component is deployed and not update those targets when later deployment actions occur.
- 3476           3. The SCA runtime **MAY** re-evaluate the target(s) for the reference dynamically as later  
3477           deployment actions occur resulting in updated reference targets which match the new  
3478           Domain configuration. How the new configuration of the reference takes place is described  
3479           by the relevant client and implementation specifications.

3480

3481       The abstract domain-level functionality for modifying the domain-level composite is as follows,  
3482       although a runtime may supply equivalent functionality in a different form:

### 3483   12.6.1 add To Domain-Level Composite

3484       This functionality adds the composite identified by a supplied URI to the Domain Level Composite.  
3485       The supplied composite URI must refer to a composite within a installed contribution. The  
3486       composite's installed contribution determines how the composite's artifacts are resolved (directly  
3487       and indirectly). The supplied composite is added to the domain composite with semantics that  
3488       correspond to the domain-level composite having an <include> statement that references the  
3489       supplied composite. All of the composite's components become *top-level* components and the  
3490       services become externally visible services (eg. they would be present in a WSDL description of  
3491       the domain).

### 3492   12.6.2 remove From Domain-Level Composite

3493       Removes from the Domain Level composite the elements corresponding to the composite  
3494       identified by a supplied composite URI. This means that the removal of the components, wires,

3495 services and references originally added to the domain level composite by the identified  
3496 composite.

### 3497 **12.6.3 get Domain-Level Composite**

3498 Returns a <composite> definition that has an <include> line for each composite that had been  
3499 added to the domain level composite. It is important to note that, in dereferencing the included  
3500 composites, any referenced artifacts must be resolved in terms of that installed composite.

### 3501 **12.6.4 get QName Definition**

3502 In order to make sense of the domain-level composite (as returned by get Domain-Level  
3503 Composite), it must be possible to get the definitions for named artifacts in the included  
3504 composites. This functionality takes the supplied URI of an installed contribution (which provides  
3505 the context), a supplied qualified name of a definition to look up, and a supplied symbol space (as  
3506 a QName, eg wsdl:PortType). The result is a single definition, in whatever form is appropriate for  
3507 that definition type.

3508 Note that this, like all the other domain-level operations, is a conceptual operation. Its capabilities  
3509 should exist in some form, but not necessarily as a service operation with exactly this signature.

---

3510 **13 Conformance**

3511 The XML schema available at the namespace URI, defined by this specification, is considered to be  
3512 authoritative and takes precedence over the XML Schema defined in the appendix of this document.

---

## 3513 A. Pseudo Schema

### 3514 A.1 ComponentType

```
3515 <?xml version="1.0" encoding="ASCII"?>
3516 <!-- Component type schema snippet -->
3517 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3518     constrainingType="QName"? >
3519
3520     <service name="xs:NCName" requires="list of xs:QName"?
3521         policySets="list of xs:QName"?>*
3522         <interface ... />
3523         <binding uri="xs:anyURI"? name="xs:NCName"?
3524             requires="list of xs:QName"?
3525             policySets="list of xs:QName"?/>*
3526         <callback?
3527             <binding ... />+
3528         </callback>
3529     </service>
3530
3531     <reference name="xs:NCName"
3532         target="list of xs:anyURI"? autowire="xs:boolean"?
3533         multiplicity="0..1 or 1..1 or 0..n or 1..n"?
3534         wiredByImpl="xs:boolean"? requires="list of xs:QName"?
3535         policySets="list of xs:QName"?>*
3536     <interface ... />
3537     <binding uri="xs:anyURI"? name="xs:NCName"?
3538         requires="list of xs:QName"?
3539         policySets="list of xs:QName"?/>*
3540     <callback?
3541         <binding ... />+
3542     </callback>
3543 </reference>
3544
3545     <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
3546         many="xs:boolean"? mustSupply="xs:boolean"?
3547         policySets="list of xs:QName"?>*
3548     default-property-value?
3549 </property>
3550
3551     <implementation requires="list of xs:QName"?
3552         policySets="list of xs:QName"?/>?
```

3553  
3554     </componentType>  
3555

## 3556 **A.2 Composite**

```
3557     <?xml version="1.0" encoding="ASCII"?>
3558     <!-- Composite schema snippet -->
3559     <composite     xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3560                   targetNamespace="xs:anyURI"
3561                   name="xs:NCName" local="xs:boolean"?
3562                   autowire="xs:boolean"? constrainingType="QName"?
3563                   requires="list of xs:QName"? policySets="list of
3564     xs:QName"?>
3565
3566         <include name="xs:QName"/>*
3567
3568         <service name="xs:NCName" promote="xs:anyURI"
3569                 requires="list of xs:QName"? policySets="list of xs:QName"?>*
3570             <interface ... />?
3571             <binding uri="xs:anyURI"? name="xs:NCName"?
3572                 requires="list of xs:QName"? policySets="list of
3573     xs:QName"?/>*
3574             <callback?
3575                 <binding uri="xs:anyURI"? name="xs:NCName"?
3576                     requires="list of xs:QName"?
3577                     policySets="list of xs:QName"?/>+
3578             </callback>
3579         </service>
3580
3581         <reference name="xs:NCName" target="list of xs:anyURI"?
3582                 promote="list of xs:anyURI" wiredByImpl="xs:boolean"?
3583                 multiplicity="0..1 or 1..1 or 0..n or 1..n"?
3584                 requires="list of xs:QName"? policySets="list of xs:QName"?>*
3585             <interface ... />?
3586             <binding uri="xs:anyURI"? name="xs:NCName"?
3587                 requires="list of xs:QName"? policySets="list of
3588     xs:QName"?/>*
3589             <callback?
3590                 <binding uri="xs:anyURI"? name="xs:NCName"?
3591                     requires="list of xs:QName"?
3592                     policySets="list of xs:QName"?/>+
3593             </callback>
3594         </reference>
3595
```

```

3596     <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
3597         many="xs:boolean"? mustSupply="xs:boolean"?>*
3598         default-property-value?
3599     </property>
3600
3601     <component name="xs:NCName" autowire="xs:boolean"?
3602         requires="list of xs:QName"? policySets="list of xs:QName"?>*
3603     <implementation ... />?
3604     <service name="xs:NCName" requires="list of xs:QName"?
3605         policySets="list of xs:QName"?>*
3606         <interface ... />?
3607         <binding uri="xs:anyURI"? name="xs:NCName"?
3608             requires="list of xs:QName"?
3609             policySets="list of xs:QName"?/>*
3610         <callback?
3611             <binding uri="xs:anyURI"? name="xs:NCName"?
3612                 requires="list of xs:QName"?
3613                 policySets="list of xs:QName"?/>+
3614         </callback>
3615     </service>
3616     <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
3617         source="xs:string"? file="xs:anyURI"? value="xs:string"?>*
3618         [<value>+ | xs:any+]?
3619     </property>
3620     <reference name="xs:NCName" target="list of xs:anyURI"?
3621         autowire="xs:boolean"? wiredByImpl="xs:boolean"?
3622         requires="list of xs:QName"? policySets="list of xs:QName"?
3623         multiplicity="0..1 or 1..1 or 0..n or 1..n"?/>*
3624     <interface ... />?
3625     <binding uri="xs:anyURI"? name="xs:NCName"?
3626         requires="list of xs:QName"?
3627         policySets="list of xs:QName"?/>*
3628     <callback?
3629         <binding uri="xs:anyURI"? name="xs:NCName"?
3630             requires="list of xs:QName"?
3631             policySets="list of xs:QName"?/>+
3632     </callback>
3633     </reference>
3634 </component>
3635
3636     <wire source="xs:anyURI" target="xs:anyURI" />*
3637
3638 </composite>

```

3639

## B. XML Schemas

3640

### B.1 sca.xsd

3641

3642

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

3643

```
<!-- (c) Copyright SCA Collaboration 2006 -->
```

3644

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

3645

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

3646

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

3647

3648

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

3649

3650

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

3651

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

3652

3653

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

3654

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

3655

3656

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

3657

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

3658

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

3659

3660

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

3661

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

3662

3663

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

3664

```
</schema>
```

3666

3667

### B.2 sca-core.xsd

3668

3669

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

3670

```
<!-- (c) Copyright SCA Collaboration 2006, 2007 -->
```

3671

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

3672

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

3673

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

3674

```
  elementFormDefault="qualified">
```

3675

3676

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

3677

```
  <complexType name="ComponentType">
```

```

3678     <sequence>
3679         <element ref="sca:implementation" minOccurs="0" maxOccurs="1"/>
3680     <choice minOccurs="0" maxOccurs="unbounded">
3681         <element name="service" type="sca:ComponentService" />
3682         <element name="reference" type="sca:ComponentReference"/>
3683         <element name="property" type="sca:Property"/>
3684     </choice>
3685     <any namespace="##other" processContents="lax" minOccurs="0"
3686         maxOccurs="unbounded"/>
3687 </sequence>
3688 <attribute name="constrainingType" type="QName" use="optional"/>
3689 <anyAttribute namespace="##other" processContents="lax"/>
3690 </complexType>
3691
3692 <element name="composite" type="sca:Composite"/>
3693 <complexType name="Composite">
3694     <sequence>
3695         <element name="include" type="anyURI" minOccurs="0"
3696             maxOccurs="unbounded"/>
3697         <choice minOccurs="0" maxOccurs="unbounded">
3698             <element name="service" type="sca:Service"/>
3699             <element name="property" type="sca:Property"/>
3700             <element name="component" type="sca:Component"/>
3701             <element name="reference" type="sca:Reference"/>
3702             <element name="wire" type="sca:Wire"/>
3703         </choice>
3704         <any namespace="##other" processContents="lax" minOccurs="0"
3705             maxOccurs="unbounded"/>
3706     </sequence>
3707     <attribute name="name" type="NCName" use="required"/>
3708     <attribute name="targetNamespace" type="anyURI" use="required"/>
3709     <attribute name="local" type="boolean" use="optional"
3710 default="false"/>
3711     <attribute name="autowire" type="boolean" use="optional"
3712 default="false"/>
3713     <attribute name="constrainingType" type="QName" use="optional"/>
3714     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3715     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3716     <anyAttribute namespace="##other" processContents="lax"/>
3717 </complexType>
3718
3719 <complexType name="Service">
3720     <sequence>

```

```

3721     <element ref="sca:interface" minOccurs="0" maxOccurs="1" />
3722     <element name="operation" type="sca:Operation" minOccurs="0"
3723         maxOccurs="unbounded" />
3724     <choice minOccurs="0" maxOccurs="unbounded">
3725         <element ref="sca:binding" />
3726         <any namespace="##other" processContents="lax"
3727             minOccurs="0" maxOccurs="unbounded" />
3728     </choice>
3729     <element ref="sca:callback" minOccurs="0" maxOccurs="1" />
3730     <any namespace="##other" processContents="lax" minOccurs="0"
3731         maxOccurs="unbounded" />
3732 </sequence>
3733 <attribute name="name" type="NCName" use="required" />
3734 <attribute name="promote" type="anyURI" use="required" />
3735 <attribute name="requires" type="sca:listOfQNames" use="optional" />
3736 <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3737 <anyAttribute namespace="##other" processContents="lax" />
3738 </complexType>
3739
3740 <element name="interface" type="sca:Interface" abstract="true" />
3741 <complexType name="Interface" abstract="true">
3742     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3743     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3744 </complexType>
3745
3746 <complexType name="Reference">
3747     <sequence>
3748         <element ref="sca:interface" minOccurs="0" maxOccurs="1" />
3749         <element name="operation" type="sca:Operation" minOccurs="0"
3750             maxOccurs="unbounded" />
3751         <choice minOccurs="0" maxOccurs="unbounded">
3752             <element ref="sca:binding" />
3753             <any namespace="##other" processContents="lax" />
3754         </choice>
3755         <element ref="sca:callback" minOccurs="0" maxOccurs="1" />
3756         <any namespace="##other" processContents="lax" minOccurs="0"
3757             maxOccurs="unbounded" />
3758     </sequence>
3759     <attribute name="name" type="NCName" use="required" />
3760     <attribute name="target" type="sca:listOfAnyURIs" use="optional"/>
3761     <attribute name="wiredByImpl" type="boolean" use="optional"
3762 default="false"/>
3763     <attribute name="multiplicity" type="sca:Multiplicity"
3764         use="optional" default="1..1" />

```

```

3765     <attribute name="promote" type="sca:listOfAnyURIs" use="required" />
3766     <attribute name="requires" type="sca:listOfQNames" use="optional" />
3767     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3768     <anyAttribute namespace="##other" processContents="lax" />
3769 </complexType>
3770
3771 <complexType name="SCAPropertyBase" mixed="true">
3772     <!-- mixed="true" to handle simple type -->
3773     <sequence>
3774         <choice minOccurs="0">
3775             <element name="value" minOccurs="1" maxOccurs="unbounded"
3776                 type="anyType"/>
3777             <any namespace="##any" processContents="lax" minOccurs="1"
3778                 maxOccurs="unbounded" />
3779             <!-- NOT an extension point; This xsd:any exists
3780                 to accept the element-based or complex type
3781                 property i.e. no element-based extension point
3782                 under "sca:property" -->
3783         </choice>
3784     </sequence>
3785 </complexType>
3786
3787 <!-- complex type for sca:property declaration -->
3788 <complexType name="Property" mixed="true">
3789     <complexContent>
3790         <extension base="sca:SCAPropertyBase">
3791             <!-- extension defines the place to hold default value -->
3792             <attribute name="name" type="NCName" use="required"/>
3793             <attribute name="value" type="xs:string" use="optional"/>
3794             <attribute name="type" type="QName" use="optional"/>
3795             <attribute name="element" type="QName" use="optional"/>
3796             <attribute name="many" type="boolean" default="false"
3797                 use="optional"/>
3798             <attribute name="mustSupply" type="boolean" default="false"
3799                 use="optional"/>
3800             <anyAttribute namespace="##other" processContents="lax"/>
3801             <!-- an extension point ; attribute-based only -->
3802         </extension>
3803     </complexContent>
3804 </complexType>
3805
3806 <complexType name="PropertyValue" mixed="true">
3807     <complexContent>

```

```

3808     <extension base="sca:SCAPropertyBase">
3809         <attribute name="name" type="NCName" use="required"/>
3810         <attribute name="value" type="xs:string" use="optional"/>
3811         <attribute name="type" type="QName" use="optional"/>
3812         <attribute name="element" type="QName" use="optional"/>
3813         <attribute name="many" type="boolean" default="false"
3814             use="optional"/>
3815         <attribute name="source" type="string" use="optional"/>
3816         <attribute name="file" type="anyURI" use="optional"/>
3817         <anyAttribute namespace="##other" processContents="lax"/>
3818         <!-- an extension point ; attribute-based only -->
3819     </extension>
3820 </complexContent>
3821 </complexType>
3822
3823 <element name="binding" type="sca:Binding" abstract="true"/>
3824 <complexType name="Binding" abstract="true">
3825     <sequence>
3826         <element name="operation" type="sca:Operation" minOccurs="0"
3827             maxOccurs="unbounded" />
3828     </sequence>
3829     <attribute name="uri" type="anyURI" use="optional"/>
3830     <attribute name="name" type="NCName" use="optional"/>
3831     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3832     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3833 </complexType>
3834
3835 <element name="bindingType" type="sca:BindingType"/>
3836 <complexType name="BindingType">
3837     <sequence minOccurs="0" maxOccurs="unbounded">
3838         <any namespace="##other" processContents="lax" />
3839     </sequence>
3840     <attribute name="type" type="QName" use="required"/>
3841     <attribute name="alwaysProvides" type="sca:listOfQNames"
3842 use="optional"/>
3843     <attribute name="mayProvide" type="sca:listOfQNames" use="optional"/>
3844     <anyAttribute namespace="##other" processContents="lax"/>
3845 </complexType>
3846
3847 <element name="callback" type="sca:Callback"/>
3848 <complexType name="Callback">
3849     <choice minOccurs="0" maxOccurs="unbounded">
3850         <element ref="sca:binding"/>

```

```

3851         <any namespace="##other" processContents="lax"/>
3852     </choice>
3853     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3854     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3855     <anyAttribute namespace="##other" processContents="lax"/>
3856 </complexType>
3857
3858 <complexType name="Component">
3859     <sequence>
3860         <element ref="sca:implementation" minOccurs="0" maxOccurs="1"/>
3861         <choice minOccurs="0" maxOccurs="unbounded">
3862             <element name="service" type="sca:ComponentService"/>
3863             <element name="reference" type="sca:ComponentReference"/>
3864             <element name="property" type="sca:PropertyValue" />
3865         </choice>
3866         <any namespace="##other" processContents="lax" minOccurs="0"
3867             maxOccurs="unbounded"/>
3868     </sequence>
3869     <attribute name="name" type="NCName" use="required"/>
3870     <attribute name="autowire" type="boolean" use="optional" />
3871     <attribute name="constrainingType" type="QName" use="optional"/>
3872     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3873     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3874     <anyAttribute namespace="##other" processContents="lax"/>
3875 </complexType>
3876
3877 <complexType name="ComponentService">
3878     <complexContent>
3879         <restriction base="sca:Service">
3880             <sequence>
3881                 <element ref="sca:interface" minOccurs="0"
3882 maxOccurs="1"/>
3883                 <element name="operation" type="sca:Operation"
3884 minOccurs="0"
3885                 maxOccurs="unbounded" />
3886                 <choice minOccurs="0" maxOccurs="unbounded">
3887                     <element ref="sca:binding"/>
3888                     <any namespace="##other" processContents="lax"
3889                         minOccurs="0" maxOccurs="unbounded"/>
3890                 </choice>
3891                 <element ref="sca:callback" minOccurs="0"
3892 maxOccurs="1"/>
3893                 <any namespace="##other" processContents="lax"
3894 minOccurs="0"

```

```

3895         maxOccurs="unbounded" />
3896     </sequence>
3897     <attribute name="name" type="NCName" use="required" />
3898     <attribute name="requires" type="sca:listOfQNames"
3899         use="optional" />
3900     <attribute name="policySets" type="sca:listOfQNames"
3901         use="optional" />
3902     <anyAttribute namespace="##other" processContents="lax" />
3903 </restriction>
3904 </complexContent>
3905 </complexType>
3906
3907 <complexType name="ComponentReference">
3908     <complexContent>
3909         <restriction base="sca:Reference">
3910             <sequence>
3911                 <element ref="sca:interface" minOccurs="0"
3912 maxOccurs="1" />
3913                 <element name="operation" type="sca:Operation"
3914 minOccurs="0"
3915         maxOccurs="unbounded" />
3916                 <choice minOccurs="0" maxOccurs="unbounded">
3917                     <element ref="sca:binding" />
3918                     <any namespace="##other" processContents="lax"
3919 />
3920                 </choice>
3921                 <element ref="sca:callback" minOccurs="0"
3922 maxOccurs="1" />
3923                 <any namespace="##other" processContents="lax"
3924 minOccurs="0"
3925         maxOccurs="unbounded" />
3926             </sequence>
3927             <attribute name="name" type="NCName" use="required" />
3928             <attribute name="autowire" type="boolean" use="optional" />
3929             <attribute name="wiredByImpl" type="boolean" use="optional"
3930                 default="false" />
3931             <attribute name="target" type="sca:listOfAnyURIs"
3932 use="optional" />
3933             <attribute name="multiplicity" type="sca:Multiplicity"
3934                 use="optional" default="1..1" />
3935             <attribute name="requires" type="sca:listOfQNames"
3936 use="optional" />
3937             <attribute name="policySets" type="sca:listOfQNames"
3938                 use="optional" />
3939             <anyAttribute namespace="##other" processContents="lax" />

```

```

3940         </restriction>
3941     </complexContent>
3942 </complexType>
3943
3944 <element name="implementation" type="sca:Implementation"
3945     abstract="true" />
3946 <complexType name="Implementation" abstract="true">
3947     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3948     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3949 </complexType>
3950
3951 <element name="implementationType" type="sca:ImplementationType"/>
3952 <complexType name="ImplementationType">
3953     <sequence minOccurs="0" maxOccurs="unbounded">
3954         <any namespace="##other" processContents="lax" />
3955     </sequence>
3956     <attribute name="type" type="QName" use="required"/>
3957     <attribute name="alwaysProvides" type="sca:listOfQNames"
3958 use="optional"/>
3959     <attribute name="mayProvide" type="sca:listOfQNames" use="optional"/>
3960     <anyAttribute namespace="##other" processContents="lax"/>
3961 </complexType>
3962
3963 <complexType name="Wire">
3964     <sequence>
3965         <any namespace="##other" processContents="lax" minOccurs="0"
3966             maxOccurs="unbounded"/>
3967     </sequence>
3968     <attribute name="source" type="anyURI" use="required"/>
3969     <attribute name="target" type="anyURI" use="required"/>
3970     <anyAttribute namespace="##other" processContents="lax"/>
3971 </complexType>
3972
3973 <element name="include" type="sca:Include"/>
3974 <complexType name="Include">
3975     <attribute name="name" type="QName"/>
3976     <anyAttribute namespace="##other" processContents="lax"/>
3977 </complexType>
3978
3979 <complexType name="Operation">
3980     <attribute name="name" type="NCName" use="required"/>
3981     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3982     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>

```

```

3983     <anyAttribute namespace="##other" processContents="lax"/>
3984 </complexType>
3985
3986 <element name="constrainingType" type="sca:ConstrainingType"/>
3987 <complexType name="ConstrainingType">
3988     <sequence>
3989         <choice minOccurs="0" maxOccurs="unbounded">
3990             <element name="service" type="sca:ComponentService"/>
3991             <element name="reference" type="sca:ComponentReference"/>
3992             <element name="property" type="sca:Property" />
3993         </choice>
3994         <any namespace="##other" processContents="lax" minOccurs="0"
3995             maxOccurs="unbounded"/>
3996     </sequence>
3997     <attribute name="name" type="NCName" use="required"/>
3998     <attribute name="targetNamespace" type="anyURI"/>
3999     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
4000     <anyAttribute namespace="##other" processContents="lax"/>
4001 </complexType>
4002
4003
4004 <simpleType name="Multiplicity">
4005     <restriction base="string">
4006         <enumeration value="0..1"/>
4007         <enumeration value="1..1"/>
4008         <enumeration value="0..n"/>
4009         <enumeration value="1..n"/>
4010     </restriction>
4011 </simpleType>
4012
4013 <simpleType name="OverrideOptions">
4014     <restriction base="string">
4015         <enumeration value="no"/>
4016         <enumeration value="may"/>
4017         <enumeration value="must"/>
4018     </restriction>
4019 </simpleType>
4020
4021 <!-- Global attribute definition for @requires to permit use of intents
4022     within WSDL documents -->
4023 <attribute name="requires" type="sca:listOfQNames"/>
4024
4025 <!-- Global attribute defintion for @endsConversation to mark operations

```

```

4026         as ending a conversation -->
4027         <attribute name="endsConversation" type="boolean" default="false"/>
4028
4029         <simpleType name="listOfQNames">
4030             <list itemType="QName"/>
4031         </simpleType>
4032
4033         <simpleType name="listOfAnyURIs">
4034             <list itemType="anyURI"/>
4035         </simpleType>
4036
4037     </schema>

```

### 4038 **B.3 sca-binding-sca.xsd**

```

4039
4040 <?xml version="1.0" encoding="UTF-8"?>
4041 <!-- (c) Copyright SCA Collaboration 2006, 2007 -->
4042 <schema xmlns="http://www.w3.org/2001/XMLSchema"
4043         targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4044         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4045         elementFormDefault="qualified">
4046
4047     <include schemaLocation="sca-core.xsd"/>
4048
4049     <element name="binding.sca" type="sca:SCABinding"
4050             substitutionGroup="sca:binding"/>
4051     <complexType name="SCABinding">
4052         <complexContent>
4053             <extension base="sca:Binding">
4054                 <sequence>
4055                     <element name="operation" type="sca:Operation"
4056 minOccurs="0"
4057                             maxOccurs="unbounded" />
4058                 </sequence>
4059                 <attribute name="uri" type="anyURI" use="optional"/>
4060                 <attribute name="name" type="QName" use="optional"/>
4061                 <attribute name="requires" type="sca:listOfQNames"
4062                             use="optional"/>
4063                 <attribute name="policySets" type="sca:listOfQNames"
4064                             use="optional"/>
4065                 <anyAttribute namespace="##other" processContents="lax"/>
4066             </extension>
4067         </complexContent>

```

```
4068     </complexType>
4069 </schema>
4070
```

## 4071 **B.4 sca-interface-java.xsd**

```
4072
4073 <?xml version="1.0" encoding="UTF-8"?>
4074 <!-- (c) Copyright SCA Collaboration 2006 -->
4075 <schema xmlns="http://www.w3.org/2001/XMLSchema"
4076     targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4077     xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4078     elementFormDefault="qualified">
4079
4080     <include schemaLocation="sca-core.xsd"/>
4081
4082     <element name="interface.java" type="sca:JavaInterface"
4083         substitutionGroup="sca:interface"/>
4084     <complexType name="JavaInterface">
4085         <complexContent>
4086             <extension base="sca:Interface">
4087                 <sequence>
4088                     <any namespace="##other" processContents="lax"
4089 minOccurs="0"
4090                     maxOccurs="unbounded"/>
4091                 </sequence>
4092                 <attribute name="interface" type="NCName" use="required"/>
4093                 <attribute name="callbackInterface" type="NCName"
4094 use="optional"/>
4095                 <anyAttribute namespace="##other" processContents="lax"/>
4096             </extension>
4097         </complexContent>
4098     </complexType>
4099 </schema>
```

## 4100 **B.5 sca-interface-wsdl.xsd**

```
4101
4102 <?xml version="1.0" encoding="UTF-8"?>
4103 <!-- (c) Copyright SCA Collaboration 2006 -->
4104 <schema xmlns="http://www.w3.org/2001/XMLSchema"
4105     targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4106     xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4107     elementFormDefault="qualified">
4108
4109     <include schemaLocation="sca-core.xsd"/>
```

```

4110
4111     <element name="interface.wsdl" type="sca:WSDLPortType"
4112           substitutionGroup="sca:interface"/>
4113     <complexType name="WSDLPortType">
4114       <complexContent>
4115         <extension base="sca:Interface">
4116           <sequence>
4117             <any namespace="##other" processContents="lax"
4118 minOccurs="0"
4119               maxOccurs="unbounded"/>
4120           </sequence>
4121           <attribute name="interface" type="anyURI" use="required"/>
4122           <attribute name="callbackInterface" type="anyURI"
4123 use="optional"/>
4124           <anyAttribute namespace="##other" processContents="lax"/>
4125         </extension>
4126       </complexContent>
4127     </complexType>
4128 </schema>

```

## 4129 **B.6 sca-implementation-java.xsd**

```

4130
4131 <?xml version="1.0" encoding="UTF-8"?>
4132 <!-- (c) Copyright SCA Collaboration 2006 -->
4133 <schema xmlns="http://www.w3.org/2001/XMLSchema"
4134       targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4135       xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4136       elementFormDefault="qualified">
4137
4138   <include schemaLocation="sca-core.xsd"/>
4139
4140   <element name="implementation.java" type="sca:JavaImplementation"
4141         substitutionGroup="sca:implementation"/>
4142   <complexType name="JavaImplementation">
4143     <complexContent>
4144       <extension base="sca:Implementation">
4145         <sequence>
4146           <any namespace="##other" processContents="lax"
4147 minOccurs="0" maxOccurs="unbounded"/>
4148         </sequence>
4149         <attribute name="class" type="NCName" use="required"/>
4150         <attribute name="requires" type="sca:listOfQNames"
4151 use="optional"/>
4152         <attribute name="policySets" type="sca:listOfQNames"

```

```

4153         use="optional"/>
4154         <anyAttribute namespace="##other" processContents="lax"/>
4155     </extension>
4156 </complexContent>
4157 </complexType>
4158 </schema>

```

## 4159 **B.7 sca-implementation-composite.xsd**

```

4160
4161 <?xml version="1.0" encoding="UTF-8"?>
4162 <!-- (c) Copyright SCA Collaboration 2006 -->
4163 <schema xmlns="http://www.w3.org/2001/XMLSchema"
4164     targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4165     xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4166     elementFormDefault="qualified">
4167
4168     <include schemaLocation="sca-core.xsd"/>
4169     <element name="implementation.composite" type="sca:SCAImplementation"
4170         substitutionGroup="sca:implementation"/>
4171     <complexType name="SCAImplementation">
4172         <complexContent>
4173             <extension base="sca:Implementation">
4174                 <sequence>
4175                     <any namespace="##other" processContents="lax"
4176 minOccurs="0"
4177                         maxOccurs="unbounded"/>
4178                 </sequence>
4179                 <attribute name="name" type="QName" use="required"/>
4180                 <attribute name="requires" type="sca:listOfQNames"
4181 use="optional"/>
4182                 <attribute name="policySets" type="sca:listOfQNames"
4183                         use="optional"/>
4184                 <anyAttribute namespace="##other" processContents="lax"/>
4185             </extension>
4186         </complexContent>
4187     </complexType>
4188 </schema>
4189

```

## 4190 **B.8 sca-definitions.xsd**

```

4191
4192 <?xml version="1.0" encoding="UTF-8"?>
4193 <!-- (c) Copyright SCA Collaboration 2006 -->

```

```

4194 <schema xmlns="http://www.w3.org/2001/XMLSchema"
4195     targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4196     xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4197     elementFormDefault="qualified">
4198
4199     <include schemaLocation="sca-core.xsd"/>
4200
4201     <element name="definitions">
4202         <complexType>
4203             <choice minOccurs="0" maxOccurs="unbounded">
4204                 <element ref="sca:intent"/>
4205                 <element ref="sca:policySet"/>
4206                 <element ref="sca:binding"/>
4207                 <element ref="sca:bindingType"/>
4208                 <element ref="sca:implementationType"/>
4209                 <any namespace="##other" processContents="lax" minOccurs="0"
4210                     maxOccurs="unbounded"/>
4211             </choice>
4212         </complexType>
4213     </element>
4214 </schema>
4215
4216

```

## 4217 **B.9 sca-binding-webservice.xsd**

4218 Is described in [the SCA Web Services Binding specification \[9\]](#)

## 4219 **B.10 sca-binding-jms.xsd**

4220 Is described in [the SCA JMS Binding specification \[11\]](#)

## 4221 **B.11 sca-policy.xsd**

4222 Is described in [the SCA Policy Framework specification \[10\]](#)

4223

## 4224 **B.12 sca-contribution.xsd**

4225

```

4226 <?xml version="1.0" encoding="UTF-8"?>
4227 <!-- (c) Copyright SCA Collaboration 2007 -->
4228 <schema xmlns="http://www.w3.org/2001/XMLSchema"
4229     targetNamespace="http://www.osea.org/xmlns/sca/1.0"
4230     xmlns:sca="http://www.osea.org/xmlns/sca/1.0"
4231     elementFormDefault="qualified">
4232
4233     <include schemaLocation="sca-core.xsd"/>
4234
4235

```

```

4236     <element name="contribution" type="sca:ContributionType"/>
4237     <complexType name="ContributionType">
4238         <sequence>
4239             <element name="deployable" type="sca:DeployableType"
4240 minOccurs="1" maxOccurs="unbounded"/>
4241             <element name="import" type="sca:ImportType" minOccurs="0"
4242 maxOccurs="unbounded"/>
4243             <element name="export" type="sca:ExportType" minOccurs="0"
4244 maxOccurs="unbounded"/>
4245             <any namespace="##other" processContents="lax" minOccurs="0"
4246 maxOccurs="unbounded"/>
4247         </sequence>
4248         <anyAttribute namespace="##other" processContents="lax"/>
4249     </complexType>
4250
4251
4252
4253     <complexType name="DeployableType">
4254         <sequence>
4255             <any namespace="##other" processContents="lax" minOccurs="0"
4256 maxOccurs="unbounded"/>
4257         </sequence>
4258         <attribute name="composite" type="QName" use="required"/>
4259         <anyAttribute namespace="##other" processContents="lax"/>
4260     </complexType>
4261
4262
4263     <complexType name="ImportType">
4264         <sequence>
4265             <any namespace="##other" processContents="lax" minOccurs="0"
4266 maxOccurs="unbounded"/>
4267         </sequence>
4268         <attribute name="namespace" type="string" use="required"/>
4269         <attribute name="location" type="anyURI" use="required"/>
4270         <anyAttribute namespace="##other" processContents="lax"/>
4271     </complexType>
4272
4273     <complexType name="ExportType">
4274         <sequence>
4275             <any namespace="##other" processContents="lax" minOccurs="0"
4276 maxOccurs="unbounded"/>
4277         </sequence>
4278         <attribute name="namespace" type="string" use="required"/>
4279         <anyAttribute namespace="##other" processContents="lax"/>
4280     </complexType>
4281
4282 </schema>
4283

```

---

4284

## C. SCA Concepts

4285

### C.1 Binding

4286  
4287  
4288

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

4289  
4290  
4291

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

4292

4293

### C.2 Component

4294  
4295  
4296  
4297  
4298  
4299

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

4300  
4301  
4302  
4303  
4304

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

4305

### C.3 Service

4306  
4307  
4308  
4309  
4310

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

4311  
4312  
4313  
4314  
4315

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

4316  
4317  
4318  
4319

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

4320

#### C.3.1 Remotable Service

4321  
4322  
4323  
4324

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

4325  
4326

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

### 4327 C.3.2 Local Service

4328 Local services are services that are designed to be only used “locally” by other implementations that are  
4329 deployed concurrently in a tightly-coupled architecture within the same operating system process.

4330 Local services may rely on by-reference calling conventions, or may assume a very fine-grained  
4331 interaction style that is incompatible with remote distribution. They may also use technology-specific data-  
4332 types.

4333 Currently a service is local only if it defined by a Java interface not marked with the @Remotable  
4334 annotation.

4335

### 4336 C.4 Reference

4337 **SCA references** represent a dependency that an implementation has on a service that is supplied by  
4338 some other implementation, where the service to be used is specified through configuration. In other  
4339 words, a reference is a service that an implementation may call during the execution of its business  
4340 function. References are typed by an interface.

4341 For composites, composite references can be accessed by components within the composite like any  
4342 service provided by a component within the composite. Composite references can be used as the targets  
4343 of wires from component references when configuring Components.

4344 A composite reference can be used to access a service such as: an SCA service provided by another  
4345 SCA composite, a Web service, a stateless session EJB, a data base stored procedure or an EIS service,  
4346 and so on. References use **bindings** to describe the access method used to their services. SCA provides  
4347 an **extensibility mechanism** that allows the introduction of new binding types to references.

4348

### 4349 C.5 Implementation

4350 An implementation is concept that is used to describe a piece of software technology such as a Java  
4351 class, BPEL process, XSLT transform, or C++ class that is used to implement one or more services in a  
4352 service-oriented application. An SCA composite is also an implementation.

4353 Implementations define points of variability including properties that can be set and settable references to  
4354 other services. The points of variability are configured by a component that uses the implementation. The  
4355 specification refers to the configurable aspects of an implementation as its **componentType**.

### 4356 C.6 Interface

4357 **Interfaces** define one or more business functions. These business functions are provided by Services  
4358 and are used by components through References. Services are defined by the Interface they implement.  
4359 SCA currently supports a number of interface type systems, for example:

- 4360 • Java interfaces
- 4361 • WSDL portTypes
- 4362 • C, C++ header files

4363

4364 SCA also provides an extensibility mechanism by which an SCA runtime can add support for additional  
4365 interface type systems.

4366 Interfaces may be **bi-directional**. A bi-directional service has service operations which must be provided  
4367 by each end of a service communication – this could be the case where a particular service requires a  
4368 “callback” interface on the client, which is calls during the process of handing service requests from the  
4369 client.

4370

## 4371 C.7 Composite

4372 An SCA composite is the basic unit of composition within an SCA Domain. An **SCA Composite** is an  
4373 assembly of Components, Services, References, and the Wires that interconnect them. Composites can  
4374 be used to contribute elements to an **SCA Domain**.

4375 A **composite** has the following characteristics:

- 4376 • It may be used as a component implementation. When used in this way, it defines a boundary for  
4377 Component visibility. Components may not be directly referenced from outside of the composite  
4378 in which they are declared.
- 4379 • It can be used to define a unit of deployment. Composites are used to contribute business logic  
4380 artifacts to an SCA domain.

4381

## 4382 C.8 Composite inclusion

4383 One composite can be used to provide part of the definition of another composite, through the process of  
4384 inclusion. This is intended to make team development of large composites easier. Included composites  
4385 are merged together into the using composite at deployment time to form a single logical composite.

4386 Composites are included into other composites through `<include.../>` elements in the using composite.  
4387 The SCA Domain uses composites in a similar way, through the deployment of composite files to a  
4388 specific location.

4389

## 4390 C.9 Property

4391 **Properties** allow for the configuration of an implementation with externally set data values. The data  
4392 value is provided through a Component, possibly sourced from the property of a containing composite.

4393 Each Property is defined by the implementation. Properties may be defined directly through the  
4394 implementation language or through annotations of implementations, where the implementation language  
4395 permits, or through a componentType file. A Property can be either a simple data type or a complex data  
4396 type. For complex data types, XML schema is the preferred technology for defining the data types.

4397

## 4398 C.10 Domain

4399 An SCA Domain represents a set of Services providing an area of Business functionality that is controlled  
4400 by a single organization. As an example, for the accounts department in a business, the SCA Domain  
4401 might cover all finance-related functions, and it might contain a series of composites dealing with specific  
4402 areas of accounting, with one for Customer accounts, another dealing with Accounts Payable.

4403 A domain specifies the instantiation, configuration and connection of a set of components, provided via  
4404 one or more composite files. The domain, like a composite, also has Services and References. Domains  
4405 also contain Wires which connect together the Components, Services and References.

4406

## 4407 C.11 Wire

4408 **SCA wires** connect **service references** to **services**.

4409 Within a composite, valid wire sources are component references and composite services. Valid wire  
4410 targets are component services and composite references.

4411 When using included composites, the sources and targets of the wires don't have to be declared in the  
4412 same composite as the composite that contains the wire. The sources and targets can be defined by  
4413 other included composites. Targets can also be external to the SCA domain.

4414

4415

---

## D. Acknowledgements

4416 The following individuals have participated in the creation of this specification and are gratefully  
4417 acknowledged:

4418 **Participants:**

4419 [Participant Name, Affiliation | Individual Member]

4420 [Participant Name, Affiliation | Individual Member]

4421

---

## E. Non-Normative Text

4423

## F. Revision History

4424 [optional; should not be included in OASIS Standards]

4425

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	<p>composite section</p> <ul style="list-style-type: none"> <li>- changed order of subsections from property, reference, service to service, reference, property</li> <li>- progressive disclosure of pseudo schemas, each section only shows what is described</li> <li>- attributes description now starts with name : type (cardinality)</li> <li>- child element description as list, each item starting with name : type (cardinality)</li> <li>- added section in appendix to contain complete pseudo schema of composite</li> </ul> <p>- moved component section after implementation section</p> <ul style="list-style-type: none"> <li>- made the ConstrainingType section a top level section</li> <li>- moved interface section to after constraining type section</li> </ul> <p>component section</p> <ul style="list-style-type: none"> <li>- added subheadings for Implementation, Service, Reference, Property</li> <li>- progressive disclosure of pseudo schemas, each section only shows what is described</li> <li>- attributes description now starts with name : type (cardinality)</li> <li>- child element description as list, each item starting with name : type (cardinality)</li> </ul> <p>implementation section</p> <ul style="list-style-type: none"> <li>- changed title to "Implementation and ComponentType"</li> <li>- moved implementation instance related stuff from implementation section to component implementation section</li> <li>- added subheadings for Service, Reference, Property, Implementation</li> <li>- progressive disclosure of pseudo schemas, each section only shows what is described</li> <li>- attributes description now starts with name : type (cardinality)</li> <li>- child element description as list, each item starting with name : type (cardinality)</li> <li>- attribute and element description still needs to be completed, all implementation statements</li> </ul>

			<p>on services, references, and properties should go here</p> <ul style="list-style-type: none"> <li>- added complete pseudo schema of componentType in appendix</li> <li>- added "Quick Tour by Sample" section, no content yet</li> <li>- added comment to introduction section that the following text needs to be added <ul style="list-style-type: none"> <li>"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."</li> </ul> </li> </ul>
3	2008-02-15	Anish Karmarkar Michael Beisiegel	<p>Incorporated resolutions from 2008 Jan f2f.</p> <ul style="list-style-type: none"> <li>- issue 9</li> <li>- issue 19</li> <li>- issue 21</li> <li>- issue 4</li> <li>- issue 1A</li> <li>- issue 27</li> <li>- in Implementation and ComponentType section added attribute and element description for service, reference, and property</li> <li>- removed comments that helped understand the initial restructuring for WD02</li> <li>- added changes for issue 43</li> <li>- added changes for issue 45, except the changes for policySet and requires attribute on property elements</li> <li>- used the NS <a href="http://docs.oasis-open.org/ns/opencsa/sca/200712">http://docs.oasis-open.org/ns/opencsa/sca/200712</a></li> <li>- updated copyright stmt</li> <li>- added wordings to make PDF normative and xml schema at the NS uri authoritative</li> </ul>
4	2008-04-22	Mike Edwards	<p>Editorial tweaks for CD01 publication:</p> <ul style="list-style-type: none"> <li>- updated URL for spec documents</li> <li>- removed comments from published CD01 version</li> <li>- removed blank pages from body of spec</li> </ul>
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: <a href="http://lists.oasis-open.org/archives/sca-assembly/200804/msg00089.html">http://lists.oasis-open.org/archives/sca-assembly/200804/msg00089.html</a></p>