



Service Component Architecture Assembly Model Specification Version 1.1

Committee Draft 01

Issue8 Proposal - v6

02 September, 2008

Deleted: 18

Deleted: 18th

Deleted: August

Deleted: March

Specification URIs:

This Version:

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

Previous Version:

Latest Version:

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

Latest Approved Version:

Technical Committee:

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

Chair(s):

Martin Chapman, Oracle
Mike Edwards, IBM

Editor(s):

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

Related work:

This specification replaces or supercedes:

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

This specification is related to:

- Service Component Architecture Policy Framework Specification Version 1.1

Declared XML Namespace(s):

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

Abstract:

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

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

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

This document describes the SCA Assembly Model, which covers

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

Status:

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

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

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

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

Notices

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

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

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

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

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

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

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

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

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

Table of Contents

1	Introduction	8
1.1	Terminology	8
1.2	Normative References	8
2	Overview.....	10
2.1	Diagram used to Represent SCA Artifacts	11
3	Quick Tour by Sample	13
4	Implementation and ComponentType	14
4.1	Component Type	14
4.1.1	Service.....	15
4.1.2	Reference	16
4.1.3	Property	18
4.1.4	Implementation	19
4.2	Example ComponentType	20
4.3	Example Implementation	20
5	Component	23
5.1	Implementation	24
5.2	Service	25
5.3	Reference	26
5.4	Property	28
5.5	Example Component	30
6	Composite	33
6.1	Service	35
6.1.1	Service Examples.....	36
6.2	Reference	37
6.2.1	Example Reference	40
6.3	Property	41
6.3.1	Property Examples	42
6.4	Wire.....	46
6.4.1	Wire Examples	48
6.4.2	Autowire.....	49
6.4.3	Autowire Examples.....	50
6.5	Using Composites as Component Implementations	53
6.5.1	Example of Composite used as a Component Implementation	55
6.6	Using Composites through Inclusion	56
6.6.1	Included Composite Examples.....	57
6.7	Composites which Include Component Implementations of Multiple Types	60
7	ConstrainingType	61
7.1	Example constrainingType	62
8	Interface.....	64
8.1	Local and Remotable Interfaces.....	65
8.2	Bidirectional Interfaces	66
8.3	Conversational Interfaces	66

8.4 SCA-Specific Aspects for WSDL Interfaces	69
Binding	70
8.5 Messages containing Data not defined in the Service Interface	72
8.6 Form of the URI of a Deployed Binding	72
8.6.1 Constructing Hierarchical URIs	72
8.6.2 Non-hierarchical URIs	73
8.6.3 Determining the URI scheme of a deployed binding.....	73
8.7 SCA Binding	74
8.7.1 Example SCA Binding	74
8.8 Web Service Binding	75
8.9 JMS Binding.....	75
9 SCA Definitions	76
10 Extension Model	77
10.1 Defining an Interface Type.....	77
10.2 Defining an Implementation Type.....	79
10.3 Defining a Binding Type.....	80
11 Packaging and Deployment	83
11.1 Domains.....	83
11.2 Contributions.....	83
11.2.1 SCA Artifact Resolution.....	84
11.2.2 SCA Contribution Metadata Document	86
11.2.3 Contribution Packaging using ZIP	87
11.3 Installed Contribution	88
11.3.1 Installed Artifact URIs	88
11.4 Operations for Contributions.....	88
11.4.1 install Contribution & update Contribution.....	89
11.4.2 add Deployment Composite & update Deployment Composite.....	89
11.4.3 remove Contribution	89
11.5 Use of Existing (non-SCA) Mechanisms for Resolving Artifacts	89
11.6 Domain-Level Composite	90
11.6.1 add To Domain-Level Composite.....	90
11.6.2 remove From Domain-Level Composite	90
11.6.3 get Domain-Level Composite.....	90
11.6.4 get QName Definition	90
12 Conformance	91
A. Pseudo Schema	92
A.1 ComponentType	92
A.2 Composite	93
B. XML Schemas	95
B.1 sca.xsd	95
B.2 sca-core.xsd	95
B.3 sca-binding-sca.xsd.....	104
B.4 sca-interface-java.xsd	104
B.5 sca-interface-wsdl.xsd.....	105
B.6 sca-implementation-java.xsd.....	106

Deleted: 85
Deleted: 86
Deleted: 87
Deleted: 87
Deleted: 87
Deleted: 88
Deleted: 88
Deleted: 88
Deleted: 88
Deleted: 89
Deleted: 89
Deleted: 89
Deleted: 89
Deleted: 90
Deleted: 91
Deleted: 91
Deleted: 92
Deleted: 94
Deleted: 94
Deleted: 94
Deleted: 103
Deleted: 103
Deleted: 104
Deleted: 105

B.7 sca-implementation-composite.xsd	<u>106</u>	Deleted: 105
B.8 sca-definitions.xsd	<u>107</u>	Deleted: 106
B.9 sca-binding-webservice.xsd	<u>108</u>	Deleted: 107
B.10 sca-binding-jms.xsd	<u>108</u>	Deleted: 107
B.11 sca-policy.xsd	<u>108</u>	Deleted: 107
C. SCA Concepts	<u>109</u>	Deleted: 108
C.1 Binding	<u>109</u>	Deleted: 108
C.2 Component	<u>109</u>	Deleted: 108
C.3 Service	<u>109</u>	Deleted: 108
C.3.1 Remotable Service	<u>109</u>	Deleted: 108
C.3.2 Local Service	<u>110</u>	Deleted: 109
C.4 Reference	<u>110</u>	Deleted: 109
C.5 Implementation	<u>110</u>	Deleted: 109
C.6 Interface	<u>110</u>	Deleted: 109
C.7 Composite	<u>111</u>	Deleted: 110
C.8 Composite inclusion	<u>111</u>	Deleted: 110
C.9 Property	<u>111</u>	Deleted: 110
C.10 Domain	<u>111</u>	Deleted: 110
C.11 Wire	<u>111</u>	Deleted: 110
D. Acknowledgements	<u>112</u>	Deleted: 111
E. Non-Normative Text	<u>113</u>	Deleted: 112
F. Revision History	<u>114</u>	Deleted: 113

1 Introduction

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

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

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

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

8 composites.

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

10

11 This specification is defined in terms of Infoset and not in terms of XML 1.0, even though the specification

12 uses XML 1.0 terminology. A mapping from XML to infoset is trivial and should be used for any non-XML

13 serializations.

14 1.1 Terminology

15 The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD

16 NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described

17 in [RFC2119].

18 1.2 Normative References

19 [RFC2119] S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*,

20 <http://www.ietf.org/rfc/rfc2119.txt>, IETF RFC 2119, March 1997.

21 [1] SCA Java Component Implementation Specification

22 SCA Java Common Annotations and APIs Specification

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

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

25

26 [2] SDO Specification

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

28

29 [3] SCA Example Code document

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

31

32 [4] JAX-WS Specification

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

34

35 [5] WS-I Basic Profile

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

37

38 [6] WS-I Basic Security Profile

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

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

63 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.

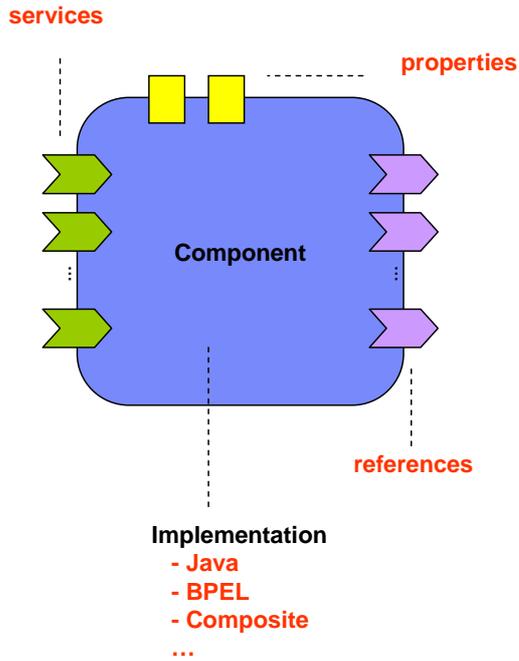
101 Composites are deployed within an **SCA Domain**. An SCA Domain typically represents a set of services
102 providing an area of business functionality that is controlled by a single organization. As an example, for
103 the accounts department in a business, the SCA Domain might cover all financial 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:



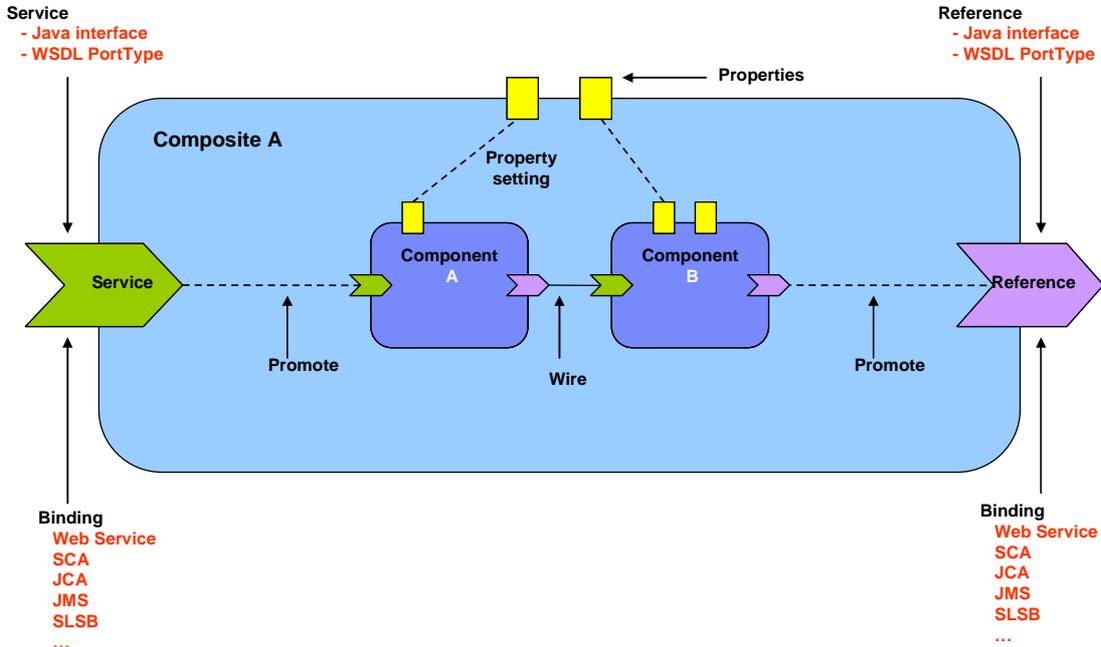
118

119 *Figure 1: SCA Component Diagram*

120

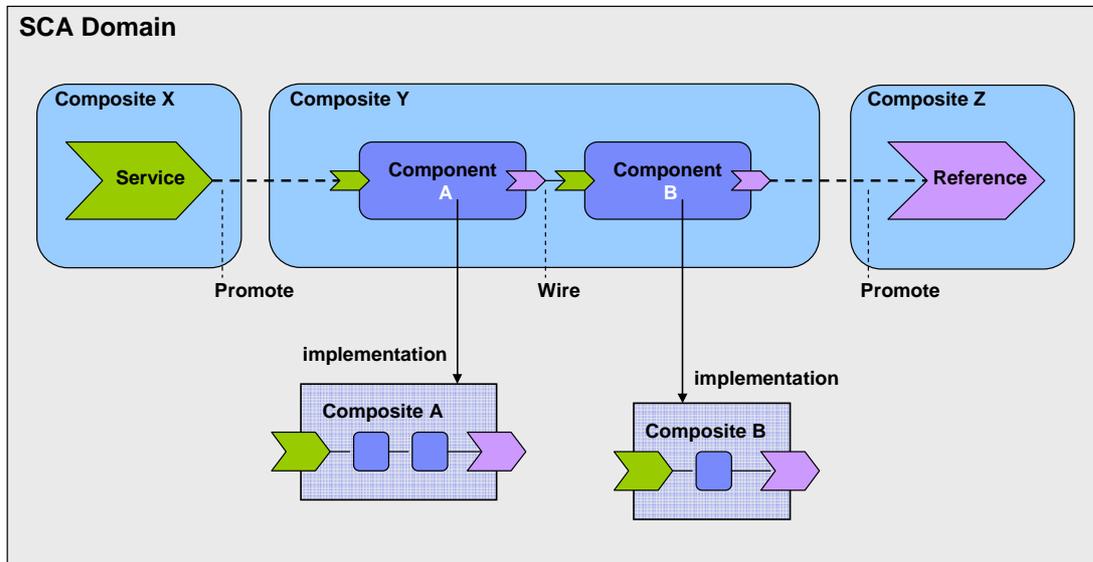
121 The following picture illustrates some of the features of a composite assembled using a set of
122 components:

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 4 Implementation and ComponentType

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

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

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

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

151 So, for example:

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

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

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

166

167 4.1 Component Type

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

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

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

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

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](#)
- 245 [\[10\]](#) for a description of this attribute.
- 246 • **policySets : QName (0..n)** - a list of policy sets. See the [Policy Framework specification](#)
- 247 [\[10\]](#) 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
- 251 operations provided by the service. The interface is described by an **interface element**
- 252 which is a child element of the service element. For details on the interface element see
- 253 [the Interface section](#).
- 254 • **binding : Binding (0..n)** - A service element has **zero or more binding elements** as
- 255 children. If the binding element is not present it defaults to <binding.sca>. Details of the
- 256 binding element are described in [the Bindings section](#). The binding, combined with any
- 257 PolicySets in effect for the binding, must satisfy the set of policy intents for the service, as
- 258 described in [the Policy Framework specification \[10\]](#).
- 259 • **callback (0..1) / binding : Binding (1..n)** - A service element has an optional **callback**
- 260 element used if the interface has a callback defined, which has one or more **binding**
- 261 elements as children. The **callback** and its binding child elements are specified if there is
- 262 a need to have binding details used to handle callbacks. If the callback element is not
- 263 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

267 another component. The reference is represented by a **reference element** which is a child of the

268 componentType element. There can be **zero or more** reference elements in a component type

269 definition. The following snippet shows the component type schema with the schema for a

270 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 ○ 1..1 – one wire can have the reference as a source
 - 301 ○ 0..1 – zero or one wire can have the reference as a source
 - 302 ○ 1..n – one or more wires can have the reference as a source
 - 303 ○ 0..n - zero 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 The **reference** element has the following **child elements**:

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

326 • **binding : Binding (0..n)** - A reference element has **zero or more binding elements** as
327 children. Details of the binding element are described in the [Bindings section](#). The binding,
328 combined with any PolicySets in effect for the binding, must satisfy the set of policy
329 intents for the reference, as described in [the Policy Framework specification \[10\]](#).

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

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

343

344 4.1.3 Property

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

349

```
350 <?xml version="1.0" encoding="ASCII"?>
351 <!-- Component type property schema snippet -->
352 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ...
353 >
354
355 <service ... />*
356 <reference ... >*
357
358 <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
359     many="xs:boolean"? mustSupply="xs:boolean"?
360     requires="list of xs:QName"?
361     policySets="list of xs:QName"?>*
362     default-property-value?
363 </property>
364
365 <implementation ... />?
366
```

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 ▪ **source : string (0..1)** - an XPath expression pointing to a property of the using
- 386 composite from which the value of this property is obtained.
- 387 ▪ **file : anyURI (0..1)** - a dereferencable URI to a file containing a value for the property.

388

389

390 4.1.4 Implementation

391 **Implementation** represents characteristics inherent to the implementation itself, in particular

392 intents and policies. See the [Policy Framework specification \[10\]](#) for a description of intents and

393 policies. The following snippet shows the component type schema with the schema for a

394 implementation child element:

395

```
396 <?xml version="1.0" encoding="ASCII"?>
397 <!-- Component type implementation schema snippet -->
398 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ...
399 >
400
401     <service ... />*
402     <reference ... >*
403     <property ... />*
404
405     <implementation requires="list of xs:QName"?
406         policySets="list of xs:QName"?/>?
407
408 </componentType>
```

Formatted: English (U.S.)

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

- 411
- **requires : QName (0..n)** - a list of policy intents. See the [Policy Framework specification \[10\]](#) for a description of this attribute.
- 412
- **policySets : QName (0..n)** - a list of policy sets. See the [Policy Framework specification \[10\]](#) for a description of this attribute.
- 413
- 414

415
416

417 4.2 Example ComponentType

418

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

422

```
423 <?xml version="1.0" encoding="ASCII"?>
424 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
425
426     <service name="MyValueService">
427         <interface.java interface="services.myvalue.MyValueService"/>
428     </service>
429
430     <reference name="customerService">
431         <interface.java interface="services.customer.CustomerService"/>
432     </reference>
433     <reference name="stockQuoteService">
434         <interface.java
435 interface="services.stockquote.StockQuoteService"/>
436     </reference>
437
438     <property name="currency" type="xsd:string">USD</property>
439
440 </componentType>
441
```

442 4.3 Example Implementation

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

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

447

```
448 package services.account;
449
450 @Remotable
451 public interface AccountService{
452
453     public AccountReport getAccountReport(String customerID);
454 }
```

455

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

459

```
460 package services.account;
```

461

```
462 import java.util.List;
```

463

```
464 import commonj.sdo.DataFactory;
```

465

```
466 import org.osoa.sca.annotations.Property;
```

```
467 import org.osoa.sca.annotations.Reference;
```

468

```
469 import services.accountdata.AccountDataService;
```

```
470 import services.accountdata.CheckingAccount;
```

```
471 import services.accountdata.SavingsAccount;
```

```
472 import services.accountdata.StockAccount;
```

```
473 import services.stockquote.StockQuoteService;
```

474

```
475 public class AccountServiceImpl implements AccountService {
```

476

```
477     @Property
```

```
478     private String currency = "USD";
```

479

```
480     @Reference
```

```
481     private AccountDataService accountDataService;
```

```
482     @Reference
```

```
483     private StockQuoteService stockQuoteService;
```

484

```
485     public AccountReport getAccountReport(String customerID) {
```

486

```
487         DataFactory dataFactory = DataFactory.INSTANCE;
```

```
488         AccountReport accountReport = (AccountReport)dataFactory.create(AccountReport.class);
```

```
489         List accountSummaries = accountReport.getAccountSummaries();
```

490

```
491         CheckingAccount checkingAccount = accountDataService.getCheckingAccount(customerID);
```

```
492         AccountSummary checkingAccountSummary =
```

```
493 (AccountSummary)dataFactory.create(AccountSummary.class);
```

```
494         checkingAccountSummary.setAccountNumber(checkingAccount.getAccountNumber());
```

```
495         checkingAccountSummary.setAccountType("checking");
```

```
496         checkingAccountSummary.setBalance(fromUSDollarToCurrency(checkingAccount.getBalance()));
```

```
497         accountSummaries.add(checkingAccountSummary);
```

498

```
499         SavingsAccount savingsAccount = accountDataService.getSavingsAccount(customerID);
```

```
500         AccountSummary savingsAccountSummary =
```

```
501 (AccountSummary)dataFactory.create(AccountSummary.class);
```

```
502         savingsAccountSummary.setAccountNumber(savingsAccount.getAccountNumber());
```

```
503         savingsAccountSummary.setAccountType("savings");
```

```
504         savingsAccountSummary.setBalance(fromUSDollarToCurrency(savingsAccount.getBalance()));
```

```
505         accountSummaries.add(savingsAccountSummary);
```

```

506
507     StockAccount stockAccount = accountDataService.getStockAccount(customerID);
508     AccountSummary stockAccountSummary =
509 (AccountSummary)dataFactory.create(AccountSummary.class);
510     stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());
511     stockAccountSummary.setAccountType("stock");
512     float balance=
513 (stockQuoteService.getQuote(stockAccount.getSymbol()))*stockAccount.getQuantity();
514     stockAccountSummary.setBalance(fromUSDollarToCurrency(balance));
515     accountSummaries.add(stockAccountSummary);
516
517     return accountReport;
518 }
519
520 private float fromUSDollarToCurrency(float value){
521
522     if (currency.equals("USD")) return value; else
523     if (currency.equals("EURO")) return value * 0.8f; else
524     return 0.0f;
525 }
526 }
527

```

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

```

530
531 <?xml version="1.0" encoding="ASCII"?>
532 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
533     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
534
535     <service name="AccountService">
536         <interface.java interface="services.account.AccountService"/>
537     </service>
538     <reference name="accountDataService">
539         <interface.java
540 interface="services.accountdata.AccountDataService"/>
541     </reference>
542     <reference name="stockQuoteService">
543         <interface.java
544 interface="services.stockquote.StockQuoteService"/>
545     </reference>
546
547     <property name="currency" type="xsd:string">USD</property>
548
549 </componentType>
550

```

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

554 5 Component

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

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

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

```
564  
565 <?xml version="1.0" encoding="UTF-8"?>  
566 <!-- Component schema snippet -->  
567 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >  
568 ...  
569   <component name="xs:NCName" autowire="xs:boolean"?  
570     requires="list of xs:QName"? policySets="list of xs:QName"?  
571     constrainingType="xs:QName"?>*  
572     <implementation ... />?  
573     <service ... />*  
574     <reference ... />*  
575     <property ... />*  
576   </component>  
577   ...  
578 </composite>  
579
```

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

- 581 • **name : NCName (1..1)** – the name of the component. The name must be unique across
582 all the components in the composite.
- 583 • **autowire : boolean (0..1)** – whether contained component references should be
584 autowired, as described in [the Autowire section](#). Default is false.
- 585 • **requires : QName (0..n)** – a list of policy intents. See the [Policy Framework specification](#)
586 [\[10\]](#) for a description of this attribute.
- 587 • **policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification](#)
588 [\[10\]](#) for a description of this attribute.
- 589 • **constrainingType : QName (0..1)** – the name of a constrainingType. When specified,
590 the set of services, references and properties of the component, plus related intents, is
591 constrained to the set defined by the constrainingType. See [the ConstrainingType Section](#)
592 for more details.

593

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

- 595 • **implementation : ComponentImplementation (0..1)** – see component
596 implementation section.

- 597 • **service** : *ComponentService (0..n)* – see component service section.
 - 598 • **reference** : *ComponentReference (0..n)* – see component reference section.
 - 599 • **property** : *ComponentProperty (0..n)* – see component property section.
- 600

601 5.1 Implementation

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

607

```
608 <?xml version="1.0" encoding="UTF-8"?>
609 <!-- Component Implementation schema snippet -->
610 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
611     ...
612     <component ... >*
613         <implementation ... />?
614         <service ... />*
615         <reference ... />*
616         <property ... />*
617     </component>
618     ...
619 </composite>
620
```

Formatted: French (France)

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

624 For example the elements **implementation.java** and **implementation.bpel** point to Java and
625 BPEL implementation types respectively. **implementation.composite** points to the use of an SCA
626 composite as an implementation. **implementation.spring** and **implementation.ejb** are used for
627 Java components written to the Spring framework and the Java EE EJB technology respectively.

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

630

```
631 <implementation.java class="services.myvalue.MyValueServiceImpl"/>
632
633 <implementation.bpel process="ans:MoneyTransferProcess"/>
634
635 <implementation.composite name="bns:MyValueComposite"/>
636
```

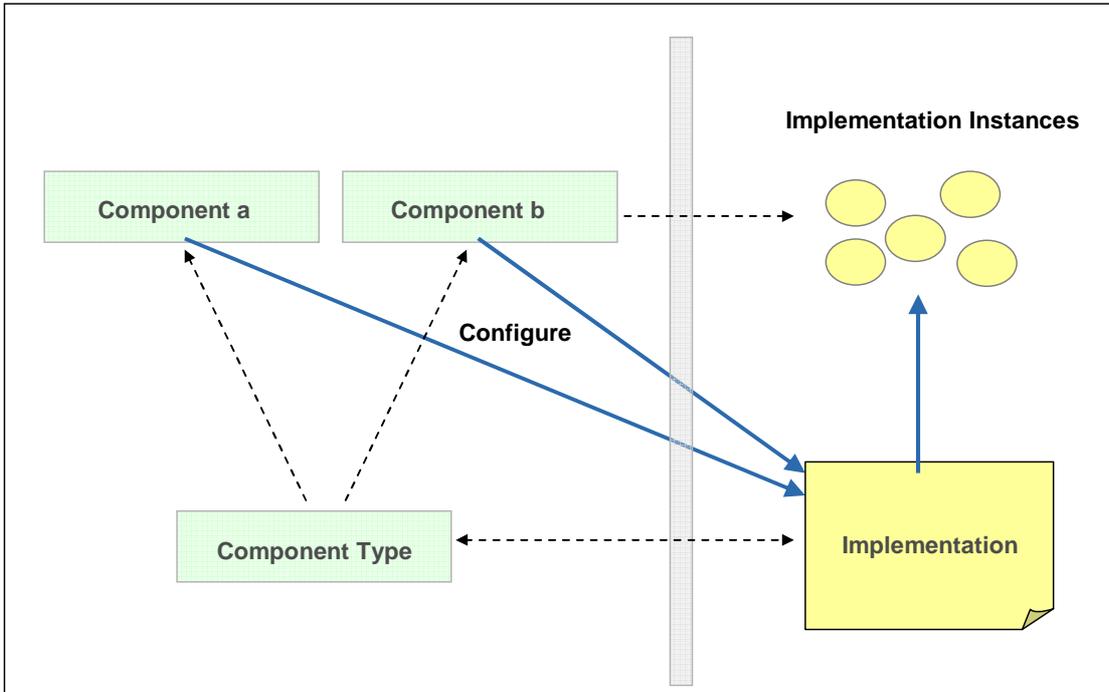
637

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

638

639 At runtime, an **implementation instance** is a specific runtime instantiation of the
640 implementation – its runtime form depends on the implementation technology used. The

641 implementation instance derives its business logic from the implementation on which it is based,
 642 but the values for its properties and references are derived from the component which configures
 643 the implementation.



644 *Figure 4: Relationship of Component and Implementation*

645

647 5.2 Service

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

652

```
653 <?xml version="1.0" encoding="UTF-8"?>
654 <!-- Component Service schema snippet -->
655 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
656   ...
657   <component ... >*
658     <implementation ... />?
659     <service name="xs:NCName" requires="list of xs:QName"?
660       policySets="list of xs:QName"?>*
661       <interface ... />?
662       <binding ... />*
663       <callback?>
```

Formatted: English (U.S.)

```

664         <binding ... />+
665     </callback>
666 </service>
667 <reference ... />*
668 <property ... />*
669 </component>
670 ...
671 </composite>
672

```

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

- 674 • **name : NCName (1..1)** - the name of the service. Has to match a name of a service
675 defined by the implementation.
- 676 • **requires : QName (0..n)** - a list of policy intents. See the [Policy Framework specification](#)
677 [\[10\]](#) for a description of this attribute.
678 Note: The effective set of policy intents for the service consists of any intents explicitly
679 stated in this requires attribute, combined with any intents specified for the service by the
680 implementation.
- 681 • **policySets : QName (0..n)** - a list of policy sets. See the [Policy Framework specification](#)
682 [\[10\]](#) for a description of this attribute.

683

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

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

704

705 5.3 Reference

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

710

```

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

```

```

712 <!-- Component Reference schema snippet -->
713 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
714 ...
715 <component ... >*
716 <implementation ... />?
717 <service ... />*
718 <reference name="xs:NCName"
719 <target="list of xs:anyURI"? autowire="xs:boolean"?
720 multiplicity="0..1 or 1..1 or 0..n or 1..n"?
721 wiredByImpl="xs:boolean"? requires="list of xs:QName"?
722 policySets="list of xs:QName"?>*
723 <interface ... />?
724 <binding uri="xs:anyURI"? requires="list of xs:QName"?
725 policySets="list of xs:QName"?/>*
726 <callback?
727 <binding ... />+
728 </callback>
729 </reference>
730 <property ... />*
731 </component>
732 ...
733 </composite>
734

```

Formatted: French (France)

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

- 736 • **name : NCName (1..1)** – the name of the reference. Has to match a name of a reference
737 defined by the implementation.
- 738 • **autowire : boolean (0..1)** – whether the reference should be autowired, as described in
739 [the Autowire section](#). Default is false.
- 740 • **requires : QName (0..n)** – a list of policy intents. See the [Policy Framework specification](#)
741 [\[10\]](#) for a description of this attribute.
742 Note: The effective set of policy intents for the reference consists of any intents explicitly
743 stated in this requires attribute, combined with any intents specified for the reference by
744 the implementation.
- 745 • **policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification](#)
746 [\[10\]](#) for a description of this attribute.
- 747 • **multiplicity : 0..1|1..1|0..n|1..n (0..1)** - defines the number of wires that can connect
748 the reference to target services. Overrides the multiplicity specified for this reference on
749 the implementation. The value can only be equal or further restrict, i.e. 0..n to 0..1 or 1..n
750 to 1..1. The multiplicity can have the following values
 - 751 ○ 1..1 – one wire can have the reference as a source
 - 752 ○ 0..1 – zero or one wire can have the reference as a source
 - 753 ○ 1..n – one or more wires can have the reference as a source
 - 754 ○ 0..n - zero or more wires can have the reference as a source
- 755 • **target : anyURI (0..n)** – a list of one or more of target service URI's, depending on
756 multiplicity setting. Each value wires the reference to a component service that resolves

757 the reference. For more details on wiring see [the section on Wires](#). Overrides any target
758 specified for this reference on the implementation.

- 759 • **wiredByImpl : boolean (0..1)** – a boolean value, "false" by default, which indicates that
760 the implementation wires this reference dynamically. If set to "true" it indicates that the
761 target of the reference is set at runtime by the implementation code (eg by the code
762 obtaining an endpoint reference by some means and setting this as the target of the
763 reference through the use of programming interfaces defined by the relevant Client and
764 Implementation specification). If "true" is set, then the reference should not be wired
765 statically within a composite, but left unwired.

766

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

- 768 • **interface : Interface (0..1)** - A reference has **zero or one interface**, which describes
769 the operations required by the reference. The interface is described by an **interface**
770 **element** which is a child element of the reference element. If no interface is specified,
771 then the interface specified for the reference by the implementation is in effect. If an
772 interface is specified it must provide a compatible superset of the interface provided by the
773 implementation, i.e. provide a superset of the operations defined by the implementation
774 for the reference. For details on the interface element see [the Interface section](#).

- 775 • **binding : Binding (0..n)** - A reference element has **zero or more binding elements** as
776 children. If no bindings are specified, then the bindings specified for the reference by the
777 implementation are in effect. If any bindings are specified, then those bindings override
778 any and all the bindings specified by the implementation. Details of the binding element
779 are described in the [Bindings section](#). The binding, combined with any PolicySets in effect
780 for the binding, must satisfy the set of policy intents for the reference, as described in [the](#)
781 [Policy Framework specification \[10\]](#).

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

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

795

796 5.4 Property

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

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

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

807

808 The form of the value of the source attribute follows the form of an XPath expression.

809 This form allows a specific property of the composite to be addressed by name. Where the
810 property is complex, the XPath expression can be extended to refer to a sub-part of the
811 complex value.

812
813 So, for example, `source="$currency"` is used to reference a property of the composite
814 called "currency", while `source="$currency/a"` references the sub-part "a" of the
815 complex composite property with the name "currency".

816

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

819

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

822

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

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

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

829

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

831

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

```
833 <!-- Component Property schema snippet -->
```

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

835

```
836 <component ... >*
```

```
837 <implementation ... />?
```

```
838 <service ... />*
```

```
839 <reference ... />*
```

```
840 <property name="xs:NCName"
```

```
841 <type="xs:QName" | element="xs:QName"?>
```

```
842 mustSupply="xs:boolean"? many="xs:boolean"?
```

```
843 source="xs:string"? file="xs:anyURI"?>*
```

```
844 <property-value?>
```

```
845 </property>
```

```
846 </component>
```

847

```
848 </composite>
```

849

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

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

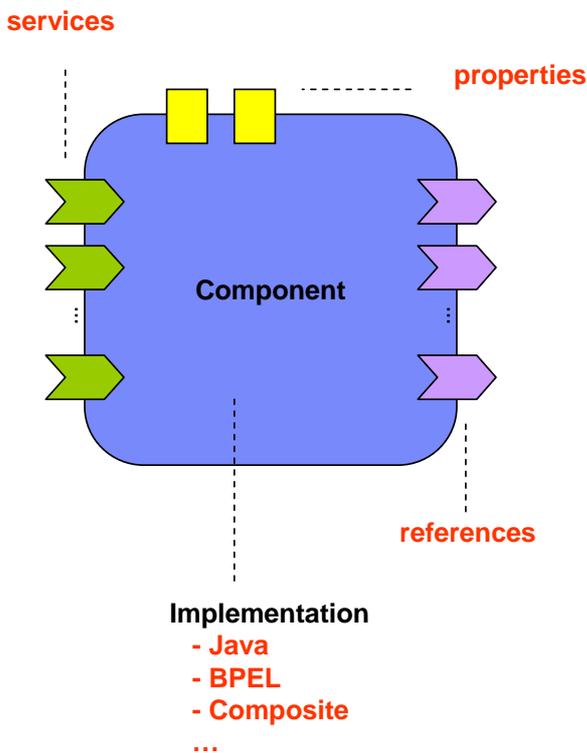
- 854 ○ **type : QName** – the type of the property defined as the qualified name of an XML
- 855 schema type
- 856 ○ **element : QName** – the type of the property defined as the qualified name of an
- 857 XML schema global element – the type is the type of the global element
- 858 ▪ **source : string (0..1)** – an XPath expression pointing to a property of the containing
- 859 composite from which the value of this component property is obtained.
- 860 ▪ **file : anyURI (0..1)** – a dereferencable URI to a file containing a value for the property
- 861 ▪ **many : boolean (0..1)** – (optional) whether the property is single-valued (false) or
- 862 multi-valued (true). Overrides the many specified for this property on the implementation.
- 863 The value can only be equal or further restrict, i.e. if the implementation specifies many
- 864 true, then the component can say false. In the case of a multi-valued property, it is
- 865 presented to the implementation as a Collection of property values.
- 866 ▪ **mustSupply : boolean (0..1)** - whether the property value must be supplied by the
- 867 component – when mustSupply="true" the component must supply a value since the
- 868 implementation has no default value for the property.
- 869

870 5.5 Example Component

871

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

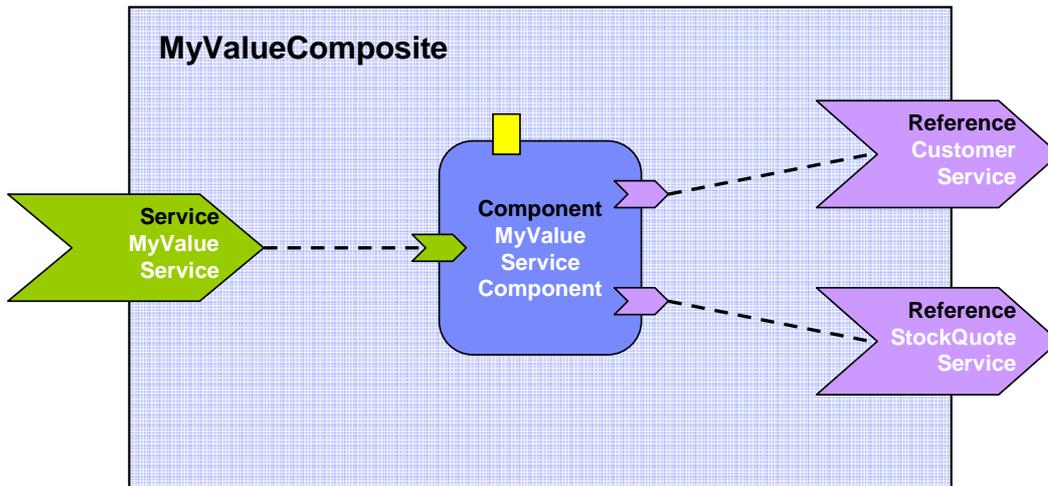
873 assembly diagram.



874

875 *Figure 5: Component symbol*

876 The following figure shows the assembly diagram for the MyValueComposite containing the
877 MyValueServiceComponent.
878



879
880
881 *Figure 6: Assembly diagram for MyValueComposite*
882

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

```
887  
888 <?xml version="1.0" encoding="ASCII"?>  
889 <!-- MyValueComposite_1 example -->  
890 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
891 targetNamespace="http://foo.com"  
892 name="MyValueComposite" >  
893  
894 <service name="MyValueService" promote="MyValueServiceComponent"/>  
895  
896 <component name="MyValueServiceComponent">  
897 <implementation.java  
898 class="services.myvalue.MyValueServiceImpl"/>  
899 <property name="currency">EURO</property>  
900 <reference name="customerService"/>  
901 <reference name="stockQuoteService"/>  
902 </component>  
903  
904 <reference name="CustomerService"  
905 promote="MyValueServiceComponent/customerService"/>
```

```

906
907     <reference name="StockQuoteService"
908         promote="MyValueServiceComponent/stockQuoteService"/>
909
910 </composite>

```

911
912 Note that the references of MyValueServiceComponent are explicitly declared only for purposes of
913 clarity – the references are defined by the MyValueServiceImpl implementation and there is no
914 need to redeclare them on the component unless the intention is to wire them or to override some
915 aspect of them.

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

```

919 <?xml version="1.0" encoding="ASCII"?>
920 <!-- MyValueComposite_2 example -->
921 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
922     targetNamespace="http://foo.com"
923     name="MyValueComposite" >
924
925     <service name="MyValueService" promote="MyValueServiceComponent"/>
926
927     <component name="MyValueServiceComponent">
928         <implementation.java
929 class="services.myvalue.MyValueServiceImpl"/>
930         <property name="currency">EURO</property>
931         <property name="currency">Yen</property>
932         <property name="currency">USDollar</property>
933         <reference name="customerService"
934             target="InternalCustomer/customerService"/>
935         <reference name="StockQuoteService"/>
936     </component>
937
938     ...
939
940     <reference name="CustomerService"
941         promote="MyValueServiceComponent/customerService"/>
942
943     <reference name="StockQuoteService"
944         promote="MyValueServiceComponent/StockQuoteService"/>
945
946 </composite>

```

947
948this assumes that the composite has another component called InternalCustomer (not shown)
949 which has a service to which the customerService reference of the MyValueServiceComponent is
950 wired as well as being promoted externally through the composite reference CustomerService.

951 6 Composite

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

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

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

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

969

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

972

```
973 <?xml version="1.0" encoding="ASCII"?>  
974 <!-- Composite schema snippet -->  
975 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
976     targetNamespace="xs:anyURI"  
977     name="xs:NCName" local="xs:boolean"?  
978     autowire="xs:boolean"? constrainingType="QName"?  
979     requires="list of xs:QName"? policySets="list of xs:QName"?>  
980  
981     <include ... />*  
982  
983     <service ... />*  
984     <reference ... />*  
985     <property ... />*  
986  
987     <component ... />*  
988  
989     <wire ... />*  
990  
991 </composite>
```

992

993

994

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

996

- **name : NCName (1..1)** – the name of the composite. The form of a composite name is an XML QName, in the namespace identified by the targetNamespace attribute.
- 997
- **targetNamespace : anyURI (0..1)** – an identifier for a target namespace into which the composite is declared
- 998
- **local : boolean (0..1)** – whether all the components within the composite must all run in the same operating system process. local="true" means that all the components must run in the same process. local="false", which is the default, means that different components within the composite may run in different operating system processes and they may even run on different nodes on a network.
- 1000
- **autowire : boolean (0..1)** – whether contained component references should be autowired, as described in [the Autowire section](#). Default is false.
- 1001
- **constrainingType : QName (0..1)** – the name of a constrainingType. When specified, the set of services, references and properties of the composite, plus related intents, is constrained to the set defined by the constrainingType. See [the ConstrainingType Section](#) for more details.
- 1002
- **requires : QName (0..n)** – a list of policy intents. See the [Policy Framework specification \[10\]](#) for a description of this attribute.
- 1003
- **policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification \[10\]](#) for a description of this attribute.
- 1004

1005

1006

1007

1008

1009

1010

1011

1012

1013

1014

1015

1016 The **composite** element has the following **child elements**:

- **service : CompositeService (0..n)** – see composite service section.
- 1017
- **reference : CompositeReference (0..n)** – see composite reference section.
- 1018
- **property : CompositeProperty (0..n)** – see composite property section.
- 1019
- **component : Component (0..n)** – see component section.
- 1020
- **wire : Wire (0..n)** – see composite wire section.
- 1021
- **include : Include (0..n)** – see composite include section
- 1022

1023

1024 Components contain configured implementations which hold the business logic of the composite.

1025 The components offer services and require references to other services. Composite services

1026 define the public services provided by the composite, which can be accessed from outside the

1027 composite. Composite references represent dependencies which the composite has on services

1028 provided elsewhere, outside the composite. Wires describe the connections between component

1029 services and component references within the composite. Included composites contribute the

1030 elements they contain to the using composite.

1031 Composite services involve the **promotion** of one service of one of the components within the

1032 composite, which means that the composite service is actually provided by one of the components

1033 within the composite. Composite references involve the **promotion** of one or more references of

1034 one or more components. Multiple component references can be promoted to the same composite

1035 reference, as long as all the component references are compatible with one another. Where

1036 multiple component references are promoted to the same composite reference, then they all share

1037 the same configuration, including the same target service(s).

1038 Composite services and composite references can use the configuration of their promoted services

1039 and references respectively (such as Bindings and Policy Sets). Alternatively composite services

1040 and composite references can override some or all of the configuration of the promoted services

1041 and references, through the configuration of bindings and other aspects of the composite service

1042 or reference.

1043 Component services and component references can be promoted to composite services and
1044 references and also be wired internally within the composite at the same time. For a reference,
1045 this only makes sense if the reference supports a multiplicity greater than 1.
1046

1047 6.1 Service

1048 The **services of a composite** are defined by promoting services defined by components
1049 contained in the composite. A component service is promoted by means of a composite **service**
1050 **element**.

1051 A composite service is represented by a **service element** which is a child of the composite
1052 element. There can be **zero or more** service elements in a composite. The following snippet
1053 shows the composite schema with the schema for a service child element:

```
1054  
1055 <?xml version="1.0" encoding="ASCII"?>  
1056 <!-- Composite Service schema snippet -->  
1057 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >  
1058   ...  
1059   <service name="xs:NCName" promote="xs:anyURI"  
1060     requires="list of xs:QName"? policySets="list of xs:QName"?>*<br>  
1061     <interface ... />?<br>  
1062     <binding ... />*<br>  
1063     <callback?>?<br>  
1064       <binding ... />+<br>  
1065     </callback><br>  
1066   </service><br>  
1067   ...<br>  
1068 </composite><br>  
1069
```

1070 The **composite service** element has the following **attributes**:

- 1071 • **name : NCName (1..1)** – the name of the service, the name MUST BE unique across all
1072 the composite services in the composite. The name of the composite service can be
1073 different from the name of the promoted component service.
- 1074 • **promote : anyURI (1..1)** – identifies the promoted service, the value is of the form
1075 <component-name>/<service-name>. The service name is optional if the target
1076 component only has one service. The same component service can be promoted by more
1077 than one composite service.
- 1078 • **requires : QName (0..n)** – a list of required policy intents. See the [Policy Framework](#)
1079 [specification \[10\]](#) for a description of this attribute. Specified **required intents** add to or
1080 further qualify the required intents defined by the promoted component service.
- 1081 • **policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification](#)
1082 [\[10\]](#) for a description of this attribute.

1083

1084 The **composite service** element has the following **child elements**, whatever is not specified is
1085 defaulted from the promoted component service.

- 1086 • **interface : Interface (0..1)** - If an **interface** is specified it must be the same or a
1087 compatible subset of the interface provided by the promoted component service, i.e.
1088 provide a subset of the operations defined by the component service. The interface is

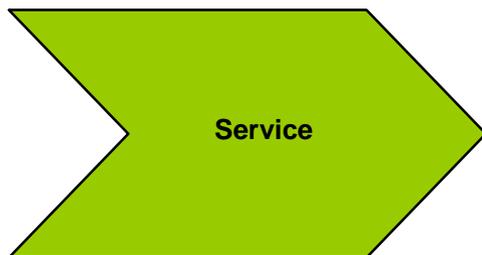
1089 described by **zero or one interface element** which is a child element of the service
 1090 element. For details on the interface element see [the Interface section](#).

- 1091 • **binding : Binding (0..n)** - If bindings are specified they **override** the bindings defined
 1092 for the promoted component service from the composite service perspective. The bindings
 1093 defined on the component service are still in effect for local wires within the composite
 1094 that target the component service. A service element has zero or more **binding elements**
 1095 as children. Details of the binding element are described in the [Bindings section](#). For more
 1096 details on wiring see [the Wiring section](#).
- 1097 • **callback (0..1) / binding : Binding (1..n)** - A service element has an optional **callback**
 1098 element used if the interface has a callback defined,, which has one or more **binding**
 1099 elements as children. The **callback** and its binding child elements are specified if there is
 1100 a need to have binding details used to handle callbacks. If the callback element is not
 1101 present, the behaviour is runtime implementation dependent.

1102

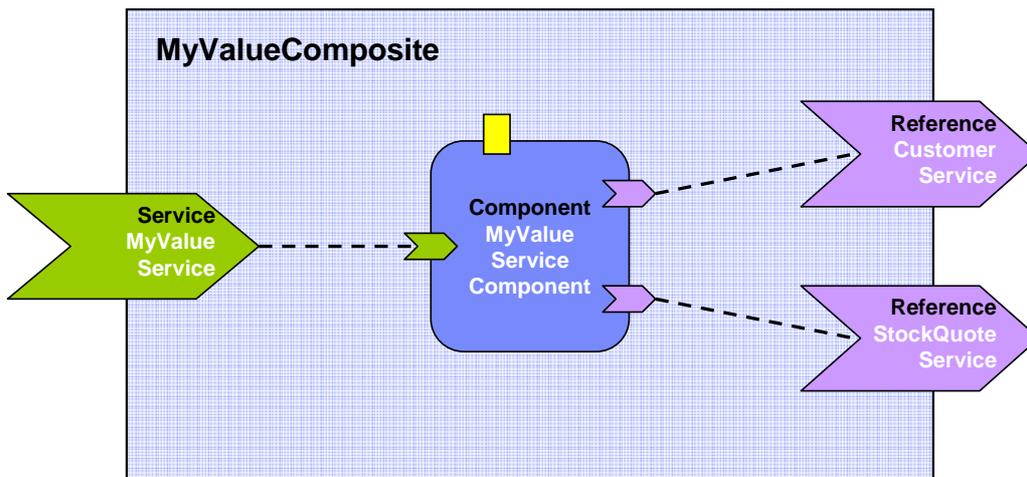
1103 6.1.1 Service Examples

1104
 1105 The following figure shows the service symbol that used to represent a service in an assembly
 1106 diagram:



1107
 1108 *Figure 7: Service symbol*

1109
 1110 The following figure shows the assembly diagram for the MyValueComposite containing the service
 1111 MyValueService.



1112
 1113 *Figure 8: MyValueComposite showing Service*

1114

1115 The following snippet shows the MyValueComposite.composite file for the MyValueComposite
1116 containing the service element for the MyValueService, which is a promote of the service offered
1117 by the MyValueServiceComponent. The name of the promoted service is omitted since
1118 MyValueServiceComponent offers only one service. The composite service MyValueService is
1119 bound using a Web service binding.

1120

```
1121 <?xml version="1.0" encoding="ASCII"?>
1122 <!-- MyValueComposite_4 example -->
1123 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1124           targetNamespace="http://foo.com"
1125           name="MyValueComposite" >
1126
1127     ...
1128
1129     <service name="MyValueService" promote="MyValueServiceComponent">
1130       <interface.java interface="services.myvalue.MyValueService"/>
1131       <binding.ws port="http://www.myvalue.org/MyValueService#
1132                 wsdl.endpoint(MyValueService/MyValueServiceSOAP)"/>
1133     </service>
1134
1135     <component name="MyValueServiceComponent">
1136       <implementation.java
1137 class="services.myvalue.MyValueServiceImpl"/>
1138       <property name="currency">EURO</property>
1139       <service name="MyValueService"/>
1140       <reference name="customerService"/>
1141       <reference name="StockQuoteService"/>
1142     </component>
1143
1144     ...
1145
1146 </composite>
```

1147

1148 6.2 Reference

1149 The **references of a composite** are defined by **promoting** references defined by components
1150 contained in the composite. Each promoted reference indicates that the component reference
1151 must be resolved by services outside the composite. A component reference is promoted using a
1152 composite **reference element**.

1153 A composite reference is represented by a **reference element** which is a child of a composite
1154 element. There can be **zero or more** reference elements in a composite. The following snippet
1155 shows the composite schema with the schema for a **reference** element.

1156

```
1157 <?xml version="1.0" encoding="ASCII"?>
```

```

1158 <!-- Composite Reference schema snippet -->
1159 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
1160 ...
1161 <reference name="xs:NCName" target="list of xs:anyURI"?
1162     promote="list of xs:anyURI" wiredByImpl="xs:boolean"?
1163     multiplicity="0..1 or 1..1 or 0..n or 1..n"?
1164     requires="list of xs:QName"? policySets="list of xs:QName"?>*
1165 <interface ... />?
1166 <binding ... />*
1167 <callback?
1168     <binding ... />+
1169 </callback>
1170 </reference>
1171 ...
1172 </composite>
1173
1174

```

1175 The **composite reference** element has the following **attributes**:

- 1176 • **name : NCName (1..1)** – the name of the reference. The name must be unique across
1177 all the composite references in the composite. The name of the composite reference can
1178 be different then the name of the promoted component reference.
- 1179 • **promote : anyURI (1..n)** – identifies one or more promoted component references. The
1180 value is a list of values of the form <component-name>/<reference-name> separated by
1181 spaces. The specification of the reference name is optional if the component has only one
1182 reference.

1183 The same component reference maybe promoted more than once, using different
1184 composite references, but only if the multiplicity defined on the component reference is
1185 0..n or 1..n. The multiplicity on the composite reference can restrict accordingly.

1186 Two or more component references may be promoted by one composite reference, but
1187 only when

- 1188 • the interfaces of the component references are the same, or if the composite
1189 reference itself declares an interface then all the component references must have
1190 interfaces which are compatible with the composite reference interface
- 1191 • the multiplicities of the component references are compatible, i.e one can be the
1192 restricted form of the another, which also means that the composite reference
1193 carries the restricted form either implicitly or explicitly
- 1194 • the intents declared on the component references must be compatible – the
1195 intents which apply to the composite reference in this case are the union of the
1196 required intents specified for each of the promoted component references. If any
1197 intents contradict (eg mutually incompatible qualifiers for a particular intent) then
1198 there is an error.

- 1199 • **requires : QName (0..n)** – a list of required policy intents. See the [Policy Framework](#)
1200 [specification \[10\]](#) for a description of this attribute. Specified **required intents** add to or
1201 further qualify the required intents defined for the promoted component reference.

- 1202 • **policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification](#)
1203 [\[10\]](#) for a description of this attribute.

- 1204 • **multiplicity : 0..1|1..1|0..n|1..n (1..1)** - Defines the number of wires that can
1205 connect the reference to target services. The multiplicity can have the following values

- 1206 o 1..1 – one wire can have the reference as a source
- 1207 o 0..1 – zero or one wire can have the reference as a source
- 1208 o 1..n – one or more wires can have the reference as a source
- 1209 o 0..n - zero or more wires can have the reference as a source

1210 The value specified for the **multiplicity** attribute has to be compatible with the multiplicity
 1211 specified on the component reference, i.e. it has to be equal or further restrict. So a
 1212 composite reference of multiplicity 0..1 or 1..1 can be used where the promoted
 1213 component reference has multiplicity 0..n and 1..n respectively. However, a composite
 1214 reference of multiplicity 0..n or 1..n cannot be used to promote a component reference of
 1215 multiplicity 0..1 or 1..1 respectively.

- 1216 • **target : anyURI (0..n)** – a list of one or more of target service URI's, depending on
 1217 multiplicity setting. Each value wires the reference to a service in a composite that uses
 1218 the composite containing the reference as an implementation for one of its components. For
 1219 more details on wiring see [the section on Wires](#).
- 1220 • **wiredByImpl : boolean (0..1)** – a boolean value, "false" by default, which indicates that
 1221 the implementation wires this reference dynamically. If set to "true" it indicates that the
 1222 target of the reference is set at runtime by the implementation code (eg by the code
 1223 obtaining an endpoint reference by some means and setting this as the target of the
 1224 reference through the use of programming interfaces defined by the relevant Client and
 1225 Implementation specification). If "true" is set, then the reference should not be wired
 1226 statically within a using composite, but left unwired.

1227
 1228 The **composite reference** element has the following **child elements**, whatever is not specified is
 1229 defaulted from the promoted component reference(s).

- 1230 • **interface : Interface (0..1)** - If an **interface** is specified it must provide an interface
 1231 which is the same or which is a compatible superset of the interface declared by the
 1232 promoted component reference, i.e. provide a superset of the operations defined by the
 1233 component for the reference. The interface is described by **zero or one interface**
 1234 **element** which is a child element of the reference element. For details on the interface
 1235 element see [the Interface section](#).
- 1236 • **binding : Binding (0..n)** - If one or more **bindings** are specified they **override** any and
 1237 all of the bindings defined for the promoted component reference from the composite
 1238 reference perspective. The bindings defined on the component reference are still in effect
 1239 for local wires within the composite that have the component reference as their source. A
 1240 reference element has zero or more **binding elements** as children. Details of the binding
 1241 element are described in the [Bindings section](#). For more details on wiring see [the section](#)
 1242 [on Wires](#).

1243 Note that a binding element may specify an endpoint which is the target of that binding. A
 1244 reference must not mix the use of endpoints specified via binding elements with target
 1245 endpoints specified via the target attribute. If the target attribute is set, then binding
 1246 elements can only list one or more binding types that can be used for the wires identified
 1247 by the target attribute. All the binding types identified are available for use on each wire
 1248 in this case. If endpoints are specified in the binding elements, each endpoint must use
 1249 the binding type of the binding element in which it is defined. In addition, each binding
 1250 element needs to specify an endpoint in this case.

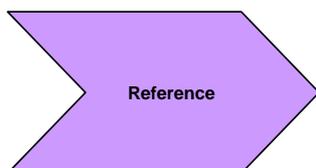
- 1251 • **callback (0..1) / binding : Binding (1..n)** - A **reference** element has an optional
 1252 **callback** element used if the interface has a callback defined, which has one or more
 1253 **binding** elements as children. The **callback** and its binding child elements are specified if
 1254 there is a need to have binding details used to handle callbacks. If the callback element is
 1255 not present, the behaviour is runtime implementation dependent.

1256

1257 **6.2.1 Example Reference**

1258

1259 The following figure shows the reference symbol that is used to represent a reference in an
1260 assembly diagram.



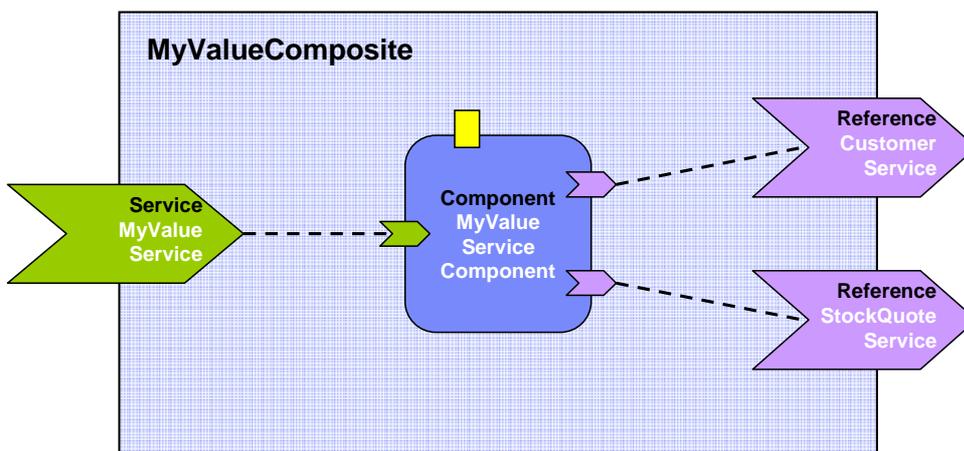
1261

1262 *Figure 9: Reference symbol*

1263

1264 The following figure shows the assembly diagram for the MyValueComposite containing the
1265 reference CustomerService and the reference StockQuoteService.

1266



1267

1268 *Figure 10: MyValueComposite showing References*

1269

1270 The following snippet shows the MyValueComposite.composite file for the MyValueComposite
1271 containing the reference elements for the CustomerService and the StockQuoteService. The
1272 reference CustomerService is bound using the SCA binding. The reference StockQuoteService
1273 is bound using the Web service binding. The endpoint addresses of the bindings can be specified, for
1274 example using the binding *uri* attribute (for details see the [Bindings](#) section), or overridden in an
1275 enclosing composite. Although in this case the reference StockQuoteService is bound to a Web
1276 service, its interface is defined by a Java interface, which was created from the WSDL portType of
1277 the target web service.

1278

```
1279 <?xml version="1.0" encoding="ASCII"?>  
1280 <!-- MyValueComposite_3 example -->  
1281 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
1282           targetNamespace="http://foo.com"  
1283           name="MyValueComposite" >
```

1284

```

1285     ...
1286
1287     <component name="MyValueServiceComponent">
1288         <implementation.java
1289         class="services.myvalue.MyValueServiceImpl"/>
1290         <property name="currency">EURO</property>
1291         <reference name="customerService"/>
1292         <reference name="StockQuoteService"/>
1293     </component>
1294
1295     <reference name="CustomerService"
1296         promote="MyValueServiceComponent/customerService">
1297         <interface.java interface="services.customer.CustomerService"/>
1298         <!-- The following forces the binding to be binding.sca whatever
1299 is -->
1300         <!-- specified by the component reference or by the underlying
1301 -->
1302         <!-- implementation
1303 -->
1304         <binding.sca/>
1305     </reference>
1306
1307     <reference name="StockQuoteService"
1308         promote="MyValueServiceComponent/StockQuoteService">
1309         <interface.java
1310         interface="services.stockquote.StockQuoteService"/>
1311         <binding.ws port="http://www.stockquote.org/StockQuoteService#
1312 wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
1313     </reference>
1314
1315     ...
1316
1317
1318 </composite>
1319

```

1320 6.3 Property

1321 **Properties** allow for the configuration of an implementation with externally set data values. A
1322 composite can declare zero or more properties. Each property has a type, which may be either
1323 simple or complex. An implementation may also define a default value for a property. Properties
1324 are configured with values in the components that use the implementation.

1325 The declaration of a property in a composite follows the form described in the following schema
1326 snippet:

```

1327
1328 <?xml version="1.0" encoding="ASCII"?>
1329 <!-- Composite Property schema snippet -->

```

```

1330 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
1331 ...
1332 <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
1333     many="xs:boolean"? mustSupply="xs:boolean"?>*
1334     default-property-value?
1335 </property>
1336 ...
1337 </composite>
1338

```

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

- 1340 ▪ **name : NCName (1..1)** - the name of the property
- 1341 ▪ one of **(1..1)**:
 - 1342 ○ **type : QName** – the type of the property - the qualified name of an XML schema
 - 1343 type
 - 1344 ○ **element : QName** – the type of the property defined as the qualified name of an
 - 1345 XML schema global element – the type is the type of the global element
- 1346 ▪ **many : boolean (0..1)** - whether the property is single-valued (false) or multi-valued
- 1347 (true). The default is **false**. In the case of a multi-valued property, it is presented to the
- 1348 implementation as a collection of property values.
- 1349 ▪ **mustSupply : boolean (0..1)** – whether the property value must be supplied by the
- 1350 component that uses the implementation – when mustSupply="true" the component must
- 1351 supply a value since the implementation has no default value for the property. A default-
- 1352 property-value should only be supplied when mustSupply="false" (the default setting for
- 1353 the mustSupply attribute), since the implication of a default value is that it is used only
- 1354 when a value is not supplied by the using component.

1355
 1356 The property element may contain an optional **default-property-value**, which provides default
 1357 value for the property. The default value must match the type declared for the property:

- 1358 ○ a string, if **type** is a simple type (must match the **type** declared)
- 1359 ○ a complex type value matching the type declared by **type**
- 1360 ○ an element matching the element named by **element**
- 1361 ○ multiple values are permitted if many="true" is specified

1363 Implementation types other than **composite** can declare properties in an implementation-
 1364 dependent form (eg annotations within a Java class), or through a property declaration of exactly
 1365 the form described above in a componentType file.

1366 Property values can be configured when an implementation is used by a component. The form of
 1367 the property configuration is shown in [the section on Components](#).

1368 6.3.1 Property Examples

1369
 1370 For the following example of Property declaration and value setting, the following complex type is
 1371 used as an example:

```

1372 <xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
1373     targetNamespace="http://foo.com/"
1374     xmlns:tns="http://foo.com/">

```

```

1375     <!-- ComplexProperty schema -->
1376     <xsd:element name="fooElement" type="MyComplexType"/>
1377     <xsd:complexType name="MyComplexType">
1378         <xsd:sequence>
1379             <xsd:element name="a" type="xsd:string"/>
1380             <xsd:element name="b" type="anyURI"/>
1381         </xsd:sequence>
1382         <attribute name="attr" type="xsd:string" use="optional"/>
1383     </xsd:complexType>
1384 </xsd:schema>
1385

```

1386 The following composite demonstrates the declaration of a property of a complex type, with a
1387 default value, plus it demonstrates the setting of a property value of a complex type within a
1388 component:

```

1389 <?xml version="1.0" encoding="ASCII"?>
1390
1391 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1392           xmlns:foo="http://foo.com"
1393           targetNamespace="http://foo.com"
1394           name="AccountServices">
1395     <!-- AccountServices Example1 -->
1396
1397     ...
1398
1399     <property name="complexFoo" type="foo:MyComplexType">
1400         <MyComplexPropertyValue xsi:type="foo:MyComplexType">
1401             <foo:a>AValue</foo:a>
1402             <foo:b>InterestingURI</foo:b>
1403         </MyComplexPropertyValue>
1404     </property>
1405
1406     <component name="AccountServiceComponent">
1407         <implementation.java class="foo.AccountServiceImpl"/>
1408         <property name="complexBar" source="$complexFoo"/>
1409         <reference name="accountDataService"
1410             target="AccountDataServiceComponent"/>
1411         <reference name="stockQuoteService" target="StockQuoteService"/>
1412     </component>
1413
1414     ...
1415
1416 </composite>

```

1417 In the declaration of the property named **complexFoo** in the composite **AccountServices**, the
1418 property is defined to be of type **foo:MyComplexType**. The namespace **foo** is declared in the

1419 composite and it references the example XSD, where MyComplexType is defined. The declaration
1420 of complexFoo contains a default value. This is declared as the content of the property element.
1421 In this example, the default value consists of the element **MyComplexPropertyValue** of type
1422 foo:MyComplexType and its two child elements <foo:a> and <foo:b>, following the definition of
1423 MyComplexType.

1424 In the component **AccountServiceComponent**, the component sets the value of the property
1425 **complexBar**, declared by the implementation configured by the component. In this case, the
1426 type of complexBar is foo:MyComplexType. The example shows that the value of the complexBar
1427 property is set from the value of the complexFoo property – the **source** attribute of the property
1428 element for complexBar declares that the value of the property is set from the value of a property
1429 of the containing composite. The value of the source attribute is **\$complexFoo**, where
1430 complexFoo is the name of a property of the composite. This value implies that the whole of the
1431 value of the source property is used to set the value of the component property.

1432 The following example illustrates the setting of the value of a property of a simple type (a string)
1433 from **part** of the value of a property of the containing composite which has a complex type:

```
1434 <?xml version="1.0" encoding="ASCII"?>
1435
1436 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1437           xmlns:foo="http://foo.com"
1438           targetNamespace="http://foo.com"
1439           name="AccountServices">
1440 <!-- AccountServices Example2 -->
1441
1442   ...
1443
1444   <property name="complexFoo" type="foo:MyComplexType">
1445     <MyComplexPropertyValue xsi:type="foo:MyComplexType">
1446       <foo:a>AValue</foo:a>
1447       <foo:b>InterestingURI</foo:b>
1448     </MyComplexPropertyValue>
1449   </property>
1450
1451   <component name="AccountServiceComponent">
1452     <implementation.java class="foo.AccountServiceImpl"/>
1453     <property name="currency" source="$complexFoo/a"/>
1454     <reference name="accountDataService"
1455               target="AccountDataServiceComponent"/>
1456     <reference name="stockQuoteService" target="StockQuoteService"/>
1457   </component>
1458
1459   ...
1460
1461 </composite>
```

1462 In this example, the component **AccountServiceComponent** sets the value of a property called
1463 **currency**, which is of type string. The value is set from a property of the composite
1464 **AccountServices** using the source attribute set to **\$complexFoo/a**. This is an XPath expression

1465 that selects the property name **complexFoo** and then selects the value of the **a** subelement of
1466 complexFoo. The "a" subelement is a string, matching the type of the currency property.

1467 Further examples of declaring properties and setting property values in a component follow:

1468 Declaration of a property with a simple type and a default value:

```
1469 <property name="SimpleTypeProperty" type="xsd:string">  
1470 MyValue  
1471 </property>
```

1472

1473 Declaration of a property with a complex type and a default value:

```
1474 <property name="complexFoo" type="foo:MyComplexType">  
1475   <MyComplexPropertyValue xsi:type="foo:MyComplexType">  
1476     <foo:a>AValue</foo:a>  
1477     <foo:b>InterestingURI</foo:b>  
1478   </MyComplexPropertyValue>  
1479 </property>
```

1480

1481 Declaration of a property with an element type:

```
1482 <property name="elementFoo" element="foo:fooElement">  
1483   <foo:fooElement>  
1484     <foo:a>AValue</foo:a>  
1485     <foo:b>InterestingURI</foo:b>  
1486   </foo:fooElement>  
1487 </property>
```

1488

1489 Property value for a simple type:

```
1490 <property name="SimpleTypeProperty">  
1491 MyValue  
1492 </property>
```

1493

1494

1495 Property value for a complex type, also showing the setting of an attribute value of the complex
1496 type:

```
1497 <property name="complexFoo">  
1498   <MyComplexPropertyValue xsi:type="foo:MyComplexType" attr="bar">  
1499     <foo:a>AValue</foo:a>  
1500     <foo:b>InterestingURI</foo:b>  
1501   </MyComplexPropertyValue>  
1502 </property>
```

1503

1504 Property value for an element type:

```
1505 <property name="elementFoo">  
1506   <foo:fooElement attr="bar">  
1507     <foo:a>AValue</foo:a>
```

```

1508         <foo:b>InterestingURI</foo:b>
1509     </foo:fooElement>
1510 </property>
1511
1512 Declaration of a property with a complex type where multiple values are supported:
1513 <property name="complexFoo" type="foo:MyComplexType" many="true"/>
1514
1515 Setting of a value for that property where multiple values are supplied:
1516 <property name="complexFoo">
1517     <MyComplexPropertyValue1 xsi:type="foo:MyComplexType" attr="bar">
1518         <foo:a>AValue</foo:a>
1519         <foo:b>InterestingURI</foo:b>
1520     </MyComplexPropertyValue1>
1521     <MyComplexPropertyValue2 xsi:type="foo:MyComplexType" attr="zing">
1522         <foo:a>BValue</foo:a>
1523         <foo:b>BoringURI</foo:b>
1524     </MyComplexPropertyValue2>
1525 </property>
1526

```

1527 6.4 Wire

1528 **SCA wires** within a composite connect **source component references** to **target component**
1529 **services**.

1530 One way of defining a wire is by **configuring a reference of a component using its target**
1531 **attribute**. The reference element is configured with the wire-target-URI of the service(s) that
1532 resolve the reference. Multiple target services are valid when the reference has a multiplicity of
1533 0..n or 1..n.

1534 An alternative way of defining a Wire is by means of a **wire element** which is a child of the
1535 composite element. There can be **zero or more** wire elements in a composite. This alternative
1536 method for defining wires is useful in circumstances where separation of the wiring from the
1537 elements the wires connect helps simplify development or operational activities. An example is
1538 where the components used to build a domain are relatively static but where new or changed
1539 applications are created regularly from those components, through the creation of new assemblies
1540 with different wiring. Deploying the wiring separately from the components allows the wiring to
1541 be created or modified with minimum effort.

1542 Note that a Wire specified via a wire element is equivalent to a wire specified via the target
1543 attribute of a reference. The rule which forbids mixing of wires specified with the target attribute
1544 with the specification of endpoints in binding subelements of the reference also applies to wires
1545 specified via separate wire elements.

1546 The following snippet shows the composite schema with the schema for the reference elements of
1547 components and composite services and the wire child element:

```

1548
1549 <?xml version="1.0" encoding="ASCII"?>
1550 <!-- Wires schema snippet -->
1551 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1552     targetNamespace="xs:anyURI"
1553     name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?

```

Formatted: German
(Germany)

Formatted: French (France)

```

1554         constrainingType="QName"?
1555         requires="list of xs:QName"? policySets="list of
1556 xs:QName"?>
1557
1558     ...
1559
1560     <wire source="xs:anyURI" target="xs:anyURI" /> *
1561
1562 </composite>

```

1565 The **reference element of a component** and the **reference element of a service** has a list of
 1566 one or more of the following **wire-target-URI** values for the target, with multiple values
 1567 separated by a space:

- 1568 • <component-name>/<service-name>
 - 1569 ○ where the target is a service of a component. The specification of the service
 1570 name is optional if the target component only has one service with a compatible
 1571 interface

1572
 1573 The **wire element** has the following attributes:

- 1574 • **source (required)** – names the source component reference. Valid URI schemes are:
 - 1575 ○ <component-name>/<reference-name>
 - 1576 ▪ where the source is a component reference. The specification of the
 1577 reference name is optional if the source component only has one reference
- 1578 • **target (required)** – names the target component service. Valid URI schemes are
 - 1579 ○ <component-name>/<service-name>
 - 1580 ▪ where the target is a service of a component. The specification of the
 1581 service name is optional if the target component only has one service with
 1582 a compatible interface

1583 For a composite used as a component implementation, wires can only link sources and targets
 1584 that are contained in the same composite (irrespective of which file or files are used to describe
 1585 the composite). Wiring to entities outside the composite is done through services and references
 1586 of the composite with wiring defined by the next higher composite.

1587 A wire may only connect a source to a target if the target implements an interface that is
 1588 compatible with the interface required by the source. The source and the target are compatible if:

- 1589 1. the source interface and the target interface MUST either both be remotable or they are
 1590 both local
- 1591 2. the operations on the target interface MUST be the same as or be a superset of the
 1592 operations in the interface specified on the source
- 1593 3. compatibility for the individual operation is defined as compatibility of the signature, that
 1594 is operation name, input types, and output types MUST BE the same.
- 1595 4. the order of the input and output types also MUST BE the same.
- 1596 5. the set of Faults and Exceptions expected by the source MUST BE the same or be a
 1597 superset of those specified by the target.
- 1598 6. other specified attributes of the two interfaces MUST match, including Scope and Callback
 1599 interface

1600 A Wire can connect between different interface languages (eg. Java interfaces and WSDL
1601 portTypes) in either direction, as long as the operations defined by the two interface types are
1602 equivalent. They are equivalent if the operation(s), parameter(s), return value(s) and
1603 faults/exceptions map to each other.

1604 Service clients cannot (portably) ask questions at runtime about additional interfaces that are
1605 provided by the implementation of the service (e.g. the result of "instance of" in Java is non
1606 portable). It is valid for an SCA implementation to have proxies for all wires, so that, for example,
1607 a reference object passed to an implementation may only have the business interface of the
1608 reference and may not be an instance of the (Java) class which is used to implement the target
1609 service, even where the interface is local and the target service is running in the same process.

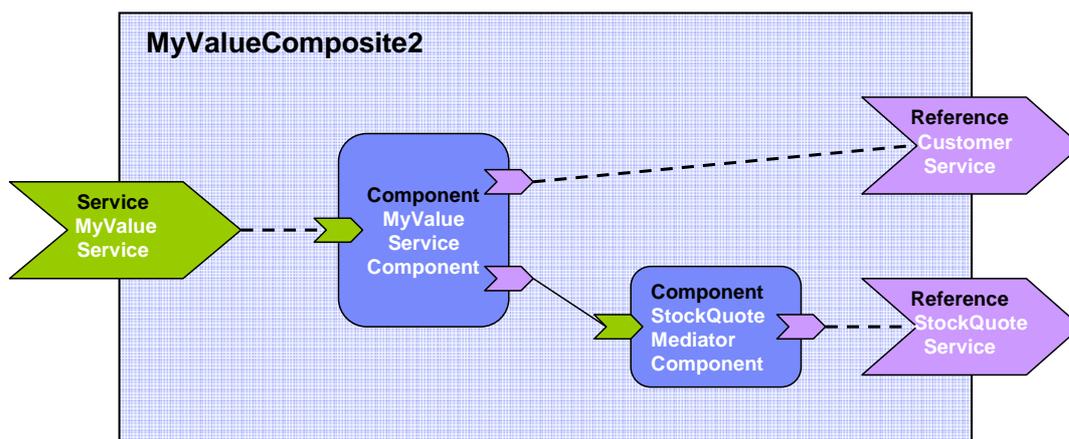
1610 **Note:** It is permitted to deploy a composite that has references that are not wired. For the case of
1611 an un-wired reference with multiplicity 1..1 or 1..n the deployment process provided by an SCA
1612 runtime SHOULD issue a warning.

1613

1614 6.4.1 Wire Examples

1615

1616 The following figure shows the assembly diagram for the MyValueComposite2 containing wires
1617 between service, components and references.



1618

1619 *Figure 11: MyValueComposite2 showing Wires*

1620

1621 The following snippet shows the MyValueComposite2.composite file for the MyValueComposite2
1622 containing the configured component and service references. The service MyValueService is wired
1623 to the MyValueServiceComponent. The MyValueServiceComponent's customerService reference is
1624 wired to the composite's CustomerService reference. The MyValueServiceComponent's
1625 stockQuoteService reference is wired to the StockQuoteMediatorComponent, which in turn has its
1626 reference wired to the StockQuoteService reference of the composite.

1627

```
1628 <?xml version="1.0" encoding="ASCII"?>  
1629 <!-- MyValueComposite Wires examples -->  
1630 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
1631           targetNamespace="http://foo.com"  
1632           name="MyValueComposite2" >
```

1633

```

1634     <service name="MyValueService" promote="MyValueServiceComponent" >
1635         <interface.java interface="services.myvalue.MyValueService" />
1636         <binding.ws port="http://www.myvalue.org/MyValueService#
1637             wsdl.endpoint(MyValueService/MyValueServiceSOAP)" />
1638     </service>
1639
1640     <component name="MyValueServiceComponent" >
1641         <implementation.java
1642 class="services.myvalue.MyValueServiceImpl" />
1643         <property name="currency">EURO</property>
1644         <service name="MyValueService" />
1645         <reference name="customerService" />
1646         <reference name="stockQuoteService"
1647             target="StockQuoteMediatorComponent" />
1648     </component>
1649
1650     <component name="StockQuoteMediatorComponent" >
1651         <implementation.java class="services.myvalue.SQMediatorImpl" />
1652         <property name="currency">EURO</property>
1653         <reference name="stockQuoteService" />
1654     </component>
1655
1656     <reference name="CustomerService"
1657         promote="MyValueServiceComponent/customerService">
1658         <interface.java interface="services.customer.CustomerService" />
1659         <binding.sca/>
1660     </reference>
1661
1662     <reference name="StockQuoteService"
1663 promote="StockQuoteMediatorComponent">
1664         <interface.java
1665 interface="services.stockquote.StockQuoteService" />
1666         <binding.ws port="http://www.stockquote.org/StockQuoteService#
1667             wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)" />
1668     </reference>
1669
1670 </composite>
1671

```

Formatted: German
(Germany)

1672 6.4.2 Autowire

1673 SCA provides a feature named **Autowire**, which can help to simplify the assembly of composites.
1674 Autowire enables component references to be automatically wired to component services which
1675 will satisfy those references, without the need to create explicit wires between the references and
1676 the services. When the autowire feature is used, a component reference which is not promoted
1677 and which is not explicitly wired to a service within a composite is automatically wired to a target

1678 service within the same composite. Autowire works by searching within the composite for a
1679 service interface which matches the interface of the references.

1680 The autowire feature is not used by default. Autowire is enabled by the setting of an autowire
1681 attribute to "true". Autowire is disabled by setting of the autowire attribute to "false" The autowire
1682 attribute can be applied to any of the following elements within a composite:

- 1683 • reference
- 1684 • component
- 1685 • composite

1686 Where an element does not have an explicit setting for the autowire attribute, it inherits the
1687 setting from its parent element. Thus a reference element inherits the setting from its containing
1688 component. A component element inherits the setting from its containing composite. Where
1689 there is no setting on any level, autowire="false" is the default.

1690 As an example, if a composite element has autowire="true" set, this means that autowiring is
1691 enabled for all component references within that composite. In this example, autowiring can be
1692 turned off for specific components and specific references through setting autowire="false" on the
1693 components and references concerned.

1694 For each component reference for which autowire is enabled, the autowire process searches within
1695 the composite for target services which are compatible with the reference. "Compatible" here
1696 means:

- 1697 • the target service interface must be a compatible superset of the reference interface (as
1698 defined in [the section on Wires](#))
- 1699 • the intents, bindings and policies applied to the service must be compatible on the
1700 reference – so that wiring the reference to the service will not cause an error due to
1701 binding and policy mismatch (see [the Policy Framework specification \[10\]](#) for details)

1702 If the search finds **more than 1** valid target service for a particular reference, the action taken
1703 depends on the multiplicity of the reference:

- 1704 • for multiplicity 0..1 and 1..1, the SCA runtime selects one of the target services in a
1705 runtime-dependent fashion and wires the reference to that target service
- 1706 • for multiplicity 0..n and 1..n, the reference is wired to all of the target services

1707 If the search finds **no** valid target services for a particular reference, the action taken depends on
1708 the multiplicity of the reference:

- 1709 • for multiplicity 0..1 and 0..n, there is no problem – no services are wired and there is no
1710 error
- 1711 • for multiplicity 1..1 and 1..n, an error is raised by the SCA runtime since the reference is
1712 intended to be wired

1713

1714 6.4.3 Autowire Examples

1715 This example demonstrates two versions of the same composite – the first version is done using
1716 explicit wires, with no autowiring used, the second version is done using autowire. In both cases
1717 the end result is the same – the same wires connect the references to the services.

1718 First, here is a diagram for the composite:

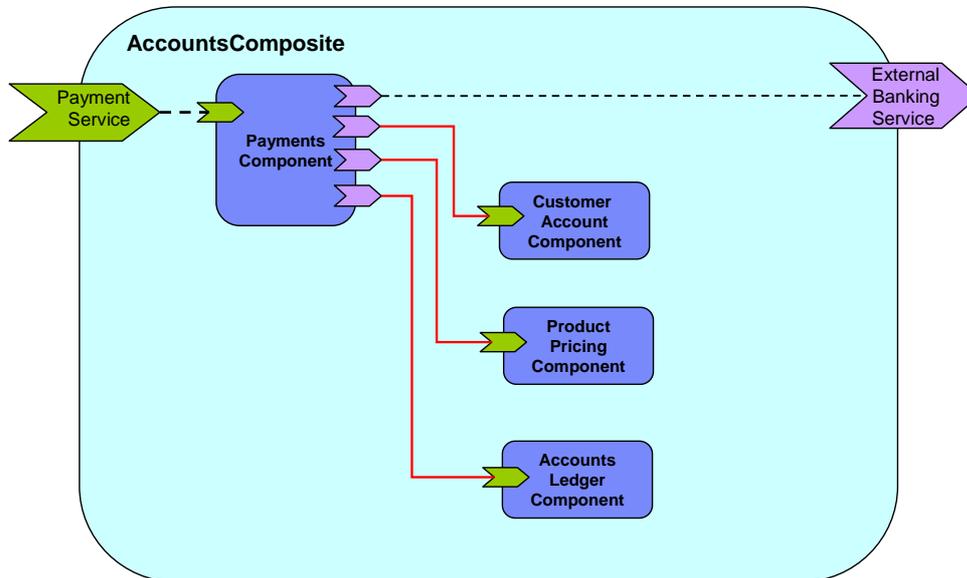


Figure 12: Example Composite for Autowire

First, the composite using explicit wires:

1719
1720
1721

1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744

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

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

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

  <component name="CustomerAccountComponent">
```

```

1745         <implementation.java class="com.foo.accounts.CustomerAccount" />
1746     </component>
1747
1748     <component name="ProductPricingComponent">
1749         <implementation.java class="com.foo.accounts.ProductPricing" />
1750     </component>
1751
1752     <component name="AccountsLedgerComponent">
1753         <implementation.composite name="foo:AccountsLedgerComposite" />
1754     </component>
1755
1756     <reference name="ExternalBankingService"
1757         promote="PaymentsComponent/ExternalBankingService" />
1758
1759 </composite>
1760

```

1761 Secondly, the composite using autowire:

```

1762 <?xml version="1.0" encoding="UTF-8"?>
1763 <!-- Autowire Example - With autowire -->
1764 <composite xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
1765     xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1766     xmlns:foo="http://foo.com"
1767     targetNamespace="http://foo.com"
1768     name="AccountComposite">
1769
1770     <service name="PaymentService" promote="PaymentsComponent">
1771         <interface.java class="com.foo.PaymentServiceInterface" />
1772     </service>
1773
1774     <component name="PaymentsComponent" autowire="true">
1775         <implementation.java class="com.foo.accounts.Payments" />
1776         <service name="PaymentService" />
1777         <reference name="CustomerAccountService" />
1778         <reference name="ProductPricingService" />
1779         <reference name="AccountsLedgerService" />
1780         <reference name="ExternalBankingService" />
1781     </component>
1782
1783     <component name="CustomerAccountComponent">
1784         <implementation.java class="com.foo.accounts.CustomerAccount" />
1785     </component>
1786
1787     <component name="ProductPricingComponent">

```

Formatted: French (France)

```

1788         <implementation.java class="com.foo.accounts.ProductPricing" />
1789     </component>
1790
1791     <component name="AccountsLedgerComponent">
1792         <implementation.composite name="foo:AccountsLedgerComposite" />
1793     </component>
1794
1795     <reference name="ExternalBankingService"
1796         promote="PaymentsComponent/ExternalBankingService" />
1797
1798 </composite>

```

1799 In this second case, autowire is set on for the PaymentsComponent and there are no explicit wires
1800 for any of its references – the wires are created automatically through autowire.

1801 **Note:** In the second example, it would be possible to omit all of the service and reference
1802 elements from the PaymentsComponent. They are left in for clarity, but if they are omitted, the
1803 component service and references still exist, since they are provided by the implementation used
1804 by the component.

1805

1806 6.5 Using Composites as Component Implementations

1807 Composites may form **component implementations** in higher-level composites – in other words
1808 the higher-level composites can have components which are implemented by composites.

1809 When a composite is used as a component implementation, it defines a boundary of visibility.
1810 Components within the composite cannot be referenced directly by the using component. The
1811 using component can only connect wires to the services and references of the used composite and
1812 set values for any properties of the composite. The internal construction of the composite is
1813 invisible to the using component.

1814 A composite used as a component implementation must also honor a **completeness contract**.
1815 The services, references and properties of the composite form a contract which is relied upon by
1816 the using component. The concept of completeness of the composite implies:

- 1817 • the composite must have at least one service or at least one reference.
1818 A component with no services and no references is not meaningful in terms of SCA, since
1819 it cannot be wired to anything – it neither provides nor consumes any services
1820
- 1821 • each service offered by the composite must be wired to a service of a component or to a
1822 composite reference.
1823 If services are left unwired, the implication is that some exception will occur at runtime if
1824 the service is invoked.

1825 The component type of a composite is defined by the set of service elements, reference elements
1826 and property elements that are the children of the composite element.

1827 Composites are used as component implementations through the use of the
1828 **implementation.composite** element as a child element of the component. The schema snippet
1829 for the implementation.composite element is:

```

1830
1831 <?xml version="1.0" encoding="ASCII"?>
1832 <!-- Composite Implementation schema snippet -->
1833 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1834     targetNamespace="xs:anyURI"

```

Formatted: English (U.S.)

```

1835         name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?
1836         constrainingType="QName"?
1837         requires="list of xs:QName"? policySets="list of
1838 xs:QName"?>
1839
1840     ...
1841
1842     <component name="xs:NCName" autowire="xs:boolean"?
1843         requires="list of xs:QName"? policySets="list of xs:QName"?>*
1844         <implementation.composite name="xs:QName"/>?
1845         <service name="xs:NCName" requires="list of xs:QName"?
1846             policySets="list of xs:QName"?>*
1847             <interface ... />?
1848             <binding uri="xs:anyURI" name="xs:QName"?
1849                 requires="list of xs:QName"
1850                 policySets="list of xs:QName"?/>*
1851             <callback?
1852                 <binding uri="xs:anyURI"? name="xs:QName"?
1853                     requires="list of xs:QName"?
1854                     policySets="list of xs:QName"?/>+
1855             </callback>
1856         </service>
1857         <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
1858             source="xs:string"? file="xs:anyURI"?>*
1859             property-value
1860         </property>
1861         <reference name="xs:NCName" target="list of xs:anyURI"?
1862             autowire="xs:boolean"? wiredByImpl="xs:boolean"?
1863             requires="list of xs:QName"? policySets="list of xs:QName"?
1864             multiplicity="0..1 or 1..1 or 0..n or 1..n"?/>*
1865         <interface ... />?
1866         <binding uri="xs:anyURI"? name="xs:QName"?
1867             requires="list of xs:QName" policySets="list of
1868 xs:QName"?/>*
1869         <callback?
1870             <binding uri="xs:anyURI"? name="xs:QName"?
1871                 requires="list of xs:QName"?
1872                 policySets="list of xs:QName"?/>+
1873         </callback>
1874         </reference>
1875     </component>
1876
1877     ...

```

1878
1879
1880
1881
1882
1883
1884

1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920

```
</composite>
```

The implementation.composite element has the following attribute:

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

6.5.1 Example of Composite used as a Component Implementation

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

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- CompositeComponent example -->
<composite xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
  xsd:schemaLocation="http://docs.oasis-open.org/ns/opencsa/sca/200712
file:/C:/Strategy/SCA/v09_osoaschemas/schemas/sca.xsd"
  xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
  targetNamespace="http://foo.com"
  xmlns:foo="http://foo.com"
  name="AccountComposite">
  <service name="AccountService" promote="AccountServiceComponent">
    <interface.java interface="services.account.AccountService"/>
    <binding.ws port="AccountService#
      wsdl.endpoint(AccountService/AccountServiceSOAP)"/>
  </service>
  <reference name="stockQuoteService"
    promote="AccountServiceComponent/StockQuoteService">
    <interface.java
interface="services.stockquote.StockQuoteService"/>
    <binding.ws
port="http://www.quickstockquote.com/StockQuoteService#
      wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
  </reference>
  <property name="currency" type="xsd:string">EURO</property>
  <component name="AccountServiceComponent">
    <implementation.composite name="foo:AccountServiceCompositel"/>
  <reference name="AccountDataService" target="AccountDataService"/>
```

Formatted: German
(Germany)

```

1921         <reference name="StockQuoteService"/>
1922
1923         <property name="currency" source="$currency"/>
1924     </component>
1925
1926     <component name="AccountDataService">
1927         <implementation.composite name="foo:AccountDataServiceComposite"/>
1928
1929         <property name="currency" source="$currency"/>
1930     </component>
1931
1932 </composite>
1933

```

1934 6.6 Using Composites through Inclusion

1935 In order to assist team development, composites may be developed in the form of multiple
1936 physical artifacts that are merged into a single logical unit. The inclusion mechanism is intended
1937 as a means to make it easier to build a large and complex composite from smaller pieces that can
1938 be worked on by different developers and assemblers, but it is assumed that the various smaller
1939 pieces are logically part of a whole. The SCA runtime MUST reject a deployed composite and raise
1940 an error where that composite, or a composite that it depends on, has an <include/> statement
1941 where the referenced included composite is not found in the same SCA contribution as the
1942 including composite.

1943 ▲ A composite is defined in an **xxx.composite** file and the composite may receive additional
1944 content through the **inclusion of other composite** files.

Deleted: ¶

1945 ▼ The semantics of included composites are that the content of the included composite is inlined into
1946 the using composite **xxx.composite** file through **include** elements in the using composite. The
1947 effect is one of **textual inclusion** – that is, the text content of the included composite is placed
1948 into the using composite in place of the include statement. The included composite element itself
1949 is discarded in this process – only its contents are included.

Deleted: ¶

1950 The composite file used for inclusion can have any contents, but always contains a single
1951 **composite** element. The composite element may contain any of the elements which are valid as
1952 child elements of a composite element, namely components, services, references, wires and
1953 includes. There is no need for the content of an included composite to be complete, so that
1954 artifacts defined within the using composite or in another associated included composite file may
1955 be referenced. For example, it is permissible to have two components in one composite file while a
1956 wire specifying one component as the source and the other as the target can be defined in a
1957 second included composite file.

1958 It is an error if the (using) composite resulting from the inclusion is invalid – for example, if there
1959 are duplicated elements in the using composite (eg. two services with the same uri contributed by
1960 different included composites), or if there are wires with non-existent source or target.

1961 The following snippet shows the partial schema for the include element.

```

1962
1963 <?xml version="1.0" encoding="UTF-8"?>
1964 <!-- Include snippet -->
1965 ▲ <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1966     targetNamespace="xs:anyURI"
1967     name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?
1968     constrainingType="QName" ?

```

Formatted: French (France)

```

1969         requires="list of xs:QName"? policySets="list of
1970 xs:QName"?>
1971
1972     ...
1973
1974     <include name="xs:QName"/>*
1975
1976     ...
1977
1978 </composite>
1979

```

1980 The include element has the following **attribute**:

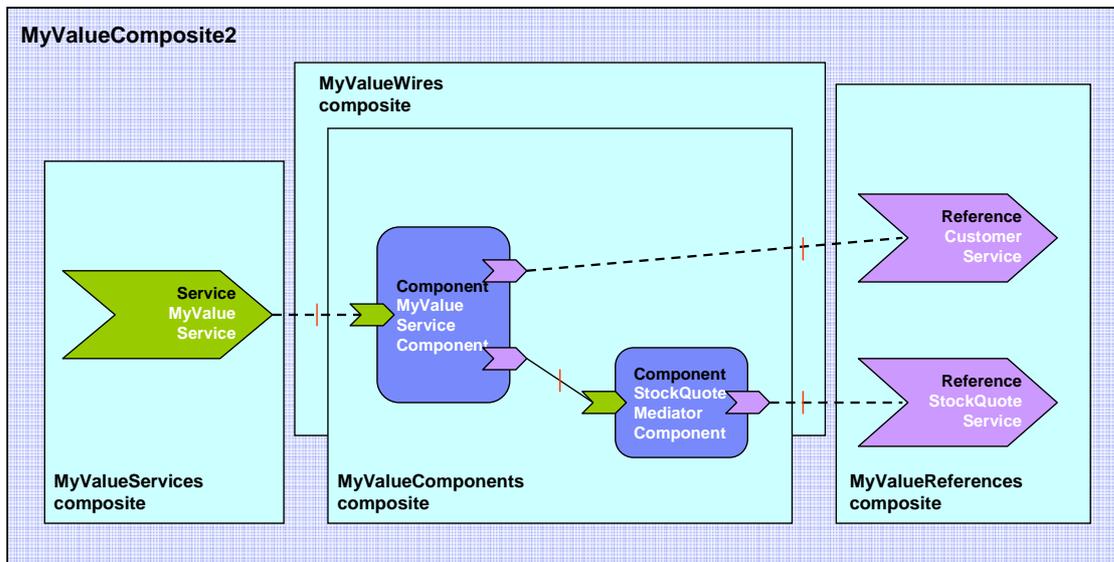
- 1981 • **name (required)** – the name of the composite that is included.

1982

1983 6.6.1 Included Composite Examples

1984

1985 The following figure shows the assembly diagram for the MyValueComposite2 containing four
1986 included composites. The **MyValueServices composite** contains the MyValueService service. The
1987 **MyValueComponents composite** contains the MyValueServiceComponent and the
1988 StockQuoteMediatorComponent as well as the wire between them. The **MyValueReferences**
1989 **composite** contains the CustomerService and StockQuoteService references. The **MyValueWires**
1990 **composite** contains the wires that connect the MyValueService service to the
1991 MyValueServiceComponent, that connect the customerService reference of the
1992 MyValueServiceComponent to the CustomerService reference, and that connect the
1993 stockQuoteService reference of the StockQuoteMediatorComponent to the StockQuoteService
1994 reference. Note that this is just one possible way of building the MyValueComposite2 from a set of
1995 included composites.



1996

1997

1998 *Figure 13 MyValueComposite2 built from 4 included composites*

1999

2000 The following snippet shows the contents of the MyValueComposite2.composite file for the
2001 MyValueComposite2 built using included composites. In this sample it only provides the name of
2002 the composite. The composite file itself could be used in a scenario using included composites to
2003 define components, services, references and wires.

2004

```
2005 <?xml version="1.0" encoding="ASCII"?>
2006 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2007           targetNamespace="http://foo.com"
2008           xmlns:foo="http://foo.com"
2009           name="MyValueComposite2" >
2010
2011     <include name="foo:MyValueServices"/>
2012     <include name="foo:MyValueComponents"/>
2013     <include name="foo:MyValueReferences"/>
2014     <include name="foo:MyValueWires"/>
2015
2016 </composite>
```

2017

2018 The following snippet shows the content of the MyValueServices.composite file.

2019

```
2020 <?xml version="1.0" encoding="ASCII"?>
2021 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2022           targetNamespace="http://foo.com"
2023           xmlns:foo="http://foo.com"
2024           name="MyValueServices" >
2025
2026     <service name="MyValueService" promote="MyValueServiceComponent" >
2027       <interface.java interface="services.myvalue.MyValueService"/>
2028       <binding.ws port="http://www.myvalue.org/MyValueService#
2029         wsdl.endpoint(MyValueService/MyValueServiceSOAP)"/>
2030     </service>
2031
2032 </composite>
```

2033

2034 The following snippet shows the content of the MyValueComponents.composite file.

2035

```
2036 <?xml version="1.0" encoding="ASCII"?>
2037 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2038           targetNamespace="http://foo.com"
2039           xmlns:foo="http://foo.com"
```

```

2040         name="MyValueComponents" >
2041
2042     <component name="MyValueServiceComponent">
2043         <implementation.java
2044 class="services.myvalue.MyValueServiceImpl"/>
2045         <property name="currency">EURO</property>
2046     </component>
2047
2048     <component name="StockQuoteMediatorComponent">
2049         <implementation.java class="services.myvalue.SQMediatorImpl"/>
2050         <property name="currency">EURO</property>
2051     </component>
2052
2053 </composite>
2054

```

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

```

2055
2056
2057 <?xml version="1.0" encoding="ASCII"?>
2058 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2059     targetNamespace="http://foo.com"
2060     xmlns:foo="http://foo.com"
2061     name="MyValueReferences" >
2062
2063     <reference name="CustomerService"
2064         promote="MyValueServiceComponent/CustomerService">
2065         <interface.java interface="services.customer.CustomerService"/>
2066         <binding.sca/>
2067     </reference>
2068
2069     <reference name="StockQuoteService"
2070 promote="StockQuoteMediatorComponent">
2071         <interface.java
2072 interface="services.stockquote.StockQuoteService"/>
2073         <binding.ws port="http://www.stockquote.org/StockQuoteService#
2074             wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
2075     </reference>
2076
2077 </composite>

```

Formatted: German
(Germany)

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

```

2078
2079
2080 <?xml version="1.0" encoding="ASCII"?>
2081 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2082     targetNamespace="http://foo.com"
2083     xmlns:foo="http://foo.com"

```

```
2084         name="MyValueWires" >
2085
2086     <wire source="MyValueServiceComponent/stockQuoteService"
2087         target="StockQuoteMediatorComponent" />
2088
2089 </composite>
```

2090 **6.7 Composites which Include Component Implementations of** 2091 **Multiple Types**

2092

2093 A Composite containing multiple components MAY have multiple component implementation types.
2094 For example, a Composite may include one component with a Java POJO as its implementation
2095 and another component with a BPEL process as its implementation.

2096

2097 7 ConstrainingType

2098 SCA allows a component, and its associated implementation, to be constrained by a
2099 **constrainingType**. The constrainingType element provides assistance in developing top-down
2100 usecases in SCA, where an architect or assembler can define the structure of a composite,
2101 including the required form of component implementations, before any of the implementations are
2102 developed.

2103 A constrainingType is expressed as an element which has services, reference and properties as
2104 child elements and which can have intents applied to it. The constrainingType is independent of
2105 any implementation. Since it is independent of an implementation it cannot contain any
2106 implementation-specific configuration information or defaults. Specifically, it cannot contain
2107 bindings, policySets, property values or default wiring information. The constrainingType is
2108 applied to a component through a constrainingType attribute on the component.

2109 A constrainingType provides the "shape" for a component and its implementation. Any component
2110 configuration that points to a constrainingType is constrained by this shape. The constrainingType
2111 specifies the services, references and properties that must be implemented. This provides the
2112 ability for the implementer to program to a specific set of services, references and properties as
2113 defined by the constrainingType. Components are therefore configured instances of
2114 implementations and are constrained by an associated constrainingType.

2115 If the configuration of the component or its implementation do not conform to the
2116 constrainingType, it is an error.

2117 A constrainingType is represented by a **constrainingType** element. The following snippet shows
2118 the pseudo-schema for the composite element.

```
2119  
2120 <?xml version="1.0" encoding="ASCII"?>  
2121 <!-- ConstrainingType schema snippet -->  
2122 <constrainingType xmlns="http://docs.oasis-  
2123 open.org/ns/opencsa/sca/200712"  
2124     targetNamespace="xs:anyURI" ?  
2125     name="xs:NCName" requires="list of xs:QName"?>  
2126  
2127  
2128     <service name="xs:NCName" requires="list of xs:QName"?>*  
2129         <interface ... />?  
2130     </service>  
2131  
2132     <reference name="xs:NCName"  
2133         multiplicity="0..1 or 1..1 or 0..n or 1..n"?  
2134         requires="list of xs:QName"?>*  
2135         <interface ... />?  
2136     </reference>  
2137  
2138     <property name="xs:NCName" (type="xs:QName" | element="xs:QName")  
2139         many="xs:boolean"? mustSupply="xs:boolean"?>*  
2140         default-property-value?  
2141     </property>
```

2142
2143
2144

```
</constrainingType>
```

2145 The constrainingType element has the following **attributes**:

- 2146 • **name (required)** – the name of the constrainingType. The form of a constrainingType
2147 name is an XML QName, in the namespace identified by the targetNamespace attribute.
- 2148 • **targetNamespace (optional)** – an identifier for a target namespace into which the
2149 constrainingType is declared
- 2150 • **requires (optional)** – a list of policy intents. See [the Policy Framework specification \[10\]](#)
2151 for a description of this attribute.

2152 ConstrainingType contains **zero or more properties, services, references**.

2153

2154 When an implementation is constrained by a constrainingType it must define all the services,
2155 references and properties specified in the corresponding constrainingType. The constraining type's
2156 references and services will have interfaces specified and may have intents specified. An
2157 implementation may contain additional services, additional optional references and additional
2158 optional properties, but cannot contain additional non-optional references or additional non-
2159 optional properties (a non-optional property is one with no default value applied).

2160 When a component is constrained by a constrainingType (via the "constrainingType" attribute),
2161 the entire componentType associated with the component and its implementation is not visible to
2162 the containing composite. The containing composite can only see a projection of the
2163 componentType associated with the component and implementation as scoped by the
2164 constrainingType of the component. For example, an additional service provided by the
2165 implementation which is not in the constrainingType associated with the component cannot be
2166 promoted by the containing composite. This requirement ensures that the constrainingType
2167 contract cannot be violated by the composite.

2168 The constrainingType can include required intents on any element. Those intents are applied to
2169 any component that uses that constrainingType. In other words, if requires="reliability" exists on
2170 a constrainingType, or its child service or reference elements, then a constrained component or its
2171 implementation must include requires="reliability" on the component or implementation or on its
2172 corresponding service or reference. Note that the component or implementation may use a
2173 qualified form of an intent specified in unqualified form in the constrainingType, but if the
2174 constrainingType uses the qualified form, then the component or implementation must also use
2175 the qualified form, otherwise there is an error.

2176 A constrainingType can be applied to an implementation. In this case, the implementation's
2177 componentType has a constrainingType attribute set to the QName of the constrainingType.

2178

2179 7.1 Example constrainingType

2180

2181 The following snippet shows the contents of the component called "MyValueServiceComponent"
2182 which is constrained by the constrainingType myns:CT. The componentType associated with the
2183 implementation is also shown.

2184

```
2185 <component name="MyValueServiceComponent" constrainingType="myns:CT">  
2186   <implementation.java class="services.myvalue.MyValueServiceImpl" />  
2187   <property name="currency">EURO</property>  
2188   <reference name="customerService" target="CustomerService">  
2189     <binding.ws ...>
```

```
2190     <reference name="StockQuoteService"
2191         target="StockQuoteMediatorComponent" />
2192 </component>
2193
2194 <constrainingType name="CT"
2195     targetNamespace="http://myns.com">
2196     <service name="MyValueService">
2197         <interface.java interface="services.myvalue.MyValueService" />
2198     </service>
2199     <reference name="customerService">
2200         <interface.java interface="services.customer.CustomerService" />
2201     </reference>
2202     <reference name="stockQuoteService">
2203         <interface.java interface="services.stockquote.StockQuoteService" />
2204     </reference>
2205     <property name="currency" type="xsd:string" />
2206 </constrainingType>
```

2207 The component MyValueServiceComponent is constrained by the constrainingType CT which
2208 means that it must provide:

- 2209 • service **MyValueService** with the interface services.myvalue.MyValueService
- 2210 • reference **customerService** with the interface services.stockquote.StockQuoteService
- 2211 • reference **stockQuoteService** with the interface services.stockquote.StockQuoteService
- 2212 • property **currency** of type xsd:string.

2213 8 Interface

2214 **Interfaces** define one or more business functions. These business functions are provided by
2215 Services and are used by References. A Service offers the business functionality of exactly one
2216 interface for use by other components. Each interface defines one or more service **operations**
2217 and each operation has zero or one **request (input) message** and zero or one **response**
2218 **(output) message**. The request and response messages may be simple types such as a string
2219 value or they may be complex types.

2220 SCA currently supports the following interface type systems:

- 2221 • Java interfaces
- 2222 • WSDL 1.1 portTypes
- 2223 • WSDL 2.0 interfaces

2224 (WSDL: [Web Services Definition Language \[8\]](#))

2225 SCA is also extensible in terms of interface types. Support for other interface type systems can be
2226 added through the extensibility mechanisms of SCA, as described in [the Extension Model section](#).

2227 The following snippet shows the schema for the Java interface element.

2228

```
2229 <interface.java interface="NCName" ... />
```

2230

2231 The interface.java element has the following attributes:

- 2232 • **interface** – the fully qualified name of the Java interface

2233

2234 The following sample shows a sample for the Java interface element.

2235

```
2236 <interface.java interface="services.stockquote.StockQuoteService"/>
```

2237

2238 Here, the Java interface is defined in the Java class file
2239 ./services/stockquote/StockQuoteService.class, where the root directory is defined by the
2240 contribution in which the interface exists.

2241 For the Java interface type system, **arguments and return** of the service methods are described
2242 using Java classes or simple Java types. [Service Data Objects \[2\]](#) are the preferred form of Java
2243 class because of their integration with XML technologies.

2244 For more information about Java interfaces, including details of SCA-specific annotations, see [the](#)
2245 [Java Client and Implementation specification \[1\]](#).

2246 The following snippet shows a sample for the WSDL portType (WSDL 1.1) or WSDL interface
2247 (WSDL 2.0) element.

2248

```
2249 <interface.wsdl interface="xs:anyURI" ... />
```

2250

2251 The interface.wsdl element has the following attributes:

- 2252 • **interface** – URI of the portType/interface with the following format
 - 2253 ○ <WSDL-namespace-URI> #wsdl.interface(<portTypeOrInterface-name>)

2254

2255 The following snippet shows a sample for the WSDL portType/interface element.

```
2256  
2257 <interface.wSDL interface="http://www.stockquote.org/StockQuoteService#  
2258                               wsdl.interface(StockQuote  
2259                               te)"/>  
2260
```

2261 For WSDL 1.1, the interface attribute points to a portType in the WSDL. For WSDL 2.0, the
2262 interface attribute points to an interface in the WSDL. For the WSDL 1.1 portType and WSDL 2.0
2263 interface type systems, arguments and return of the service operations are described using XML
2264 schema.

2265
2266

2267 8.1 Local and Remotable Interfaces

2268 A remotable service is one which may be called by a client which is running in an operating system
2269 process different from that of the service itself (this also applies to clients running on different
2270 machines from the service). Whether a service of a component implementation is remotable is
2271 defined by the interface of the service. In the case of Java this is defined by adding the
2272 **@Remotable** annotation to the Java interface (see [Client and Implementation Model Specification
2273 for Java](#)). WSDL defined interfaces are always remotable.

2274

2275 The style of remotable interfaces is typically **coarse grained** and intended for **loosely coupled**
2276 interactions. Remotable service Interfaces MUST NOT make use of **method or operation**
2277 **overloading**.

2278
2279 Independent of whether the remotable service is called remotely from outside the process where
2280 the service runs or from another component running in the same process, the data exchange
2281 semantics are **by-value**.

2282 Implementations of remotable services may modify input messages (parameters) during or after
2283 an invocation and may modify return messages (results) after the invocation. If a remotable
2284 service is called locally or remotely, the SCA container is responsible for making sure that no
2285 modification of input messages or post-invocation modifications to return messages are seen by
2286 the caller.

2287 Here is a snippet which shows an example of a remotable java interface:

2288

```
2289 package services.hello;  
2290  
2291 @Remotable  
2292 public interface HelloService {  
2293  
2294     String hello(String message);  
2295 }  
2296
```

2296

2297 It is possible for the implementation of a remotable service to indicate that it can be called using
2298 by-reference data exchange semantics when it is called from a component in the same process.
2299 This can be used to improve performance for service invocations between components that run in
2300 the same process. This can be done using the **@AllowsPassByReference** annotation (see the [Java
2301 Client and Implementation Specification](#)).

2302

2303 A service typed by a local interface can only be called by clients that are running in the same
2304 process as the component that implements the local service. Local services cannot be published
2305 via remotable services of a containing composite. In the case of Java a local service is defined by a
2306 Java interface definition without a **@Remotable** annotation.

2307

2308 The style of local interfaces is typically **fine grained** and intended for **tightly coupled**
2309 interactions. Local service interfaces can make use of **method or operation overloading**.

2310 The data exchange semantic for calls to services typed by local interfaces is **by-reference**.

2311

2312 8.2 Bidirectional Interfaces

2313 The relationship of a business service to another business service is often peer-to-peer, requiring
2314 a two-way dependency at the service level. In other words, a business service represents both a
2315 consumer of a service provided by a partner business service and a provider of a service to the
2316 partner business service. This is especially the case when the interactions are based on
2317 asynchronous messaging rather than on remote procedure calls. The notion of **bidirectional**
2318 **interfaces** is used in SCA to directly model peer-to-peer bidirectional business service
2319 relationships.

2320 An interface element for a particular interface type system must allow the specification of an
2321 optional callback interface. If a callback interface is specified SCA refers to the interface as a whole
2322 as a bidirectional interface.

2323 The following snippet shows the interface element defined using Java interfaces with an optional
2324 callbackInterface attribute.

2325

```
2326 <interface.java          interface="services.invoicing.ComputePrice"  
2327                       callbackInterface="services.invoicing.InvoiceCallback"/>
```

2328

2329 If a service is defined using a bidirectional interface element then its implementation implements
2330 the interface, and its implementation uses the callback interface to converse with the client that
2331 called the service interface.

2332

2333 If a reference is defined using a bidirectional interface element, the client component
2334 implementation using the reference calls the referenced service using the interface. The client
2335 must provide an implementation of the callback interface.

2336 Callbacks may be used for both remotable and local services. Either both interfaces of a
2337 bidirectional service MUST be remotable, or both MUST be local. A bidirectional service MUST NOT
2338 mix local and remote services.

2339

2340 8.3 Conversational Interfaces

2341

2342 Services sometimes cannot easily be defined so that each operation stands alone and is
2343 completely independent of the other operations of the same service. Instead, there is a sequence
2344 of operations that must be called in order to achieve some higher level goal. SCA calls this
2345 sequence of operations a **conversation**. If the service uses a bidirectional interface, the
2346 conversation may include both operations and callbacks.

2347

2348 Such conversational services are typically managed by using conversation identifiers that are
2349 either (1) part of the application data (message parts or operation parameters) or 2)
2350 communicated separately from application data (possibly in headers). SCA introduces the concept
2351 of *conversational interfaces* for describing the interface contract for conversational services of the
2352 second form above. With this form, it is possible for the runtime to automatically manage the
2353 conversation, with the help of an appropriate binding specified at deployment. SCA does not
2354 standardize any aspect of conversational services that are maintained using application data.
2355 Such services are neither helped nor hindered by SCA's conversational service support.

2356
2357 Conversational services typically involve state data that relates to the conversation that is taking
2358 place. The creation and management of the state data for a conversation has a significant impact
2359 on the development of both clients and implementations of conversational services.

2360
2361 Traditionally, application developers who have needed to write conversational services have been
2362 required to write a lot of plumbing code. They need to:

- 2363
- 2364 - choose or define a protocol to communicate conversational (correlation) information
2365 between the client & provider
 - 2366 - route conversational messages in the provider to a machine that can handle that
2367 conversation, while handling concurrent data access issues
 - 2368 - write code in the client to use/encode the conversational information
 - 2369 - maintain state that is specific to the conversation, sometimes persistently and
2370 transactionally, both in the implementation and the client.

2371
2372 SCA makes it possible to divide the effort associated with conversational services between a
2373 number of roles:

- 2374 - Application Developer: Declares that a service interface is conversational (leaving the
2375 details of the protocol up to the binding). Uses lifecycle semantics, APIs or other
2376 programmatic mechanisms (as defined by the implementation-type being used) to
2377 manage conversational state.
- 2378 - Application Assembler: chooses a binding that can support conversations
- 2379 - Binding Provider: implements a protocol that can pass conversational information with
2380 each operation request/response.
- 2381 - Implementation-Type Provider: defines APIs and/or other programmatic mechanisms for
2382 application developers to access conversational information. Optionally implements
2383 instance lifecycle semantics that automatically manage implementation state based on
2384 the binding's conversational information.

2385
2386 This specification requires interfaces to be marked as conversational by means of a policy intent
2387 with the name **"conversational"**. The form of the marking of this intent depends on the
2388 interface type. Note that it is also possible for a service or a reference to set the conversational
2389 intent when using an interface which is not marked with the conversational intent. This can be
2390 useful when reusing an existing interface definition that does not contain SCA information.

2391 The meaning of the conversational intent is that both the client and the provider of the interface
2392 may assume that messages (in either direction) will be handled as part of an ongoing conversation
2393 without depending on identifying information in the body of the message (i.e. in parameters of the
2394 operations). In effect, the conversation interface specifies a high-level abstract protocol that must
2395 be satisfied by any actual binding/policy combination used by the service.

2396 Examples of binding/policy combinations that support conversational interfaces are:

2397 - Web service binding with a WS-RM policy

2398 - Web service binding with a WS-Addressing policy

2399 - Web service binding with a WS-Context policy

2400 - JMS binding with a conversation policy that uses the JMS correlationID header

2401

2402 Conversations occur between one client and one target service. Consequently, requests originating
2403 from one client to multiple target conversational services will result in multiple conversations. For
2404 example, if a client A calls services B and C, both of which implement conversational interfaces,
2405 two conversations result, one between A and B and another between A and C. Likewise, requests
2406 flowing through multiple implementation instances will result in multiple conversations. For
2407 example, a request flowing from A to B and then from B to C will involve two conversations (A and
2408 B, B and C). In the previous example, if a request was then made from C to A, a third
2409 conversation would result (and the implementation instance for A would be different from the one
2410 making the original request).

2411 Invocation of any operation of a conversational interface MAY start a conversation. The decision on
2412 whether an operation would start a conversation depends on the component's implementation and
2413 its implementation type. Implementation types MAY support components with conversational
2414 services. If an implementation type does provide this support, it must provide a mechanism for
2415 determining when a new conversation should be used for an operation (for example, in Java, the
2416 conversation is new on the first use of an injected reference; in BPEL, the conversation is new
2417 when the client's partnerLink comes into scope).

2418

2419 One or more operations in a conversational interface may be annotated with an *endsConversation*
2420 annotation (the mechanism for annotating the interface depends on the interface type). Where an
2421 interface is **bidirectional**, operations may also be annotated in this way on operations of a
2422 callback interface. When a conversation ending operation is called, it indicates to both the client
2423 and the service provider that the conversation is complete. Any subsequent attempts to call an
2424 operation or a callback operation associated with the same conversation will generate a
2425 `sca:ConversationViolation` fault.

2426 A `sca:ConversationViolation` fault is thrown when one of the following errors occur:

2427 - A message is received for a particular conversation, after the conversation has ended

2428 - The conversation identification is invalid (not unique, out of range, etc.)

2429 - The conversation identification is not present in the input message of the operation that
2430 ends the conversation

2431 - The client or the service attempts to send a message in a conversation, after the
2432 conversation has ended

2433 This fault is named within the SCA namespace standard prefix "sca", which corresponds to URI
2434 <http://docs.oasis-open.org/ns/opencsa/sca/200712>.

2435 The lifecycle of resources and the association between unique identifiers and conversations are
2436 determined by the service's implementation type and may not be directly affected by the
2437 "endConversation" annotation. For example, a **WS-BPEL** process may outlive most of the
2438 conversations that it is involved in.

2439 Although conversational interfaces do not require that any identifying information be passed as
2440 part of the body of messages, there is conceptually an identity associated with the conversation.
2441 Individual implementations types MAY provide an API to access the ID associated with the
2442 conversation, although no assumptions may be made about the structure of that identifier.
2443 Implementation types MAY also provide a means to set the conversation ID by either the client or
2444 the service provider, although the operation may only be supported by some binding/policy
2445 combinations.

2446

2447 Implementation-type specifications are encouraged to define and provide conversational instance
2448 lifecycle management for components that implement conversational interfaces. However,
2449 implementations may also manage the conversational state manually.

2450

2451 8.4 SCA-Specific Aspects for WSDL Interfaces

2452 There are a number of aspects that SCA applies to interfaces in general, such as marking them
2453 **conversational**. These aspects apply to the interfaces themselves, rather than their use in a
2454 specific place within SCA. There is thus a need to provide appropriate ways of marking the
2455 interface definitions themselves, which go beyond the basic facilities provided by the interface
2456 definition language.

2457 For WSDL interfaces, there is an extension mechanism that permits additional information to be
2458 included within the WSDL document. SCA takes advantage of this extension mechanism. In order
2459 to use the SCA extension mechanism, the SCA namespace ([http://docs.oasis-
2460 open.org/ns/opencsa/sca/200712](http://docs.oasis-open.org/ns/opencsa/sca/200712)) must be declared within the WSDL document.

2461 First, SCA defines a global attribute in the SCA namespace which provides a mechanism to attach
2462 policy intents - **@requires**. The definition of this attribute is as follows:

```
2463 <attribute name="requires" type="sca:listOfQNames" />
```

2464

```
2465 <simpleType name="listOfQNames">
```

```
2466 <list itemType="QName" />
```

```
2467 </simpleType>
```

2468 The @requires attribute can be applied to WSDL Port Type elements (WSDL 1.1) and to WSDL
2469 Interface elements (WSDL 2.0). The attribute contains one or more intent names, as defined by
2470 the [Policy Framework specification \[10\]](#). Any service or reference that uses an interface with
2471 required intents implicitly adds those intents to its own @requires list.

2472 To specify that a WSDL interface is conversational, the following attribute setting is used on either
2473 the WSDL Port Type or WSDL Interface:

```
2474 requires="conversational"
```

2475 SCA defines an **endsConversation** attribute that is used to mark specific operations within a
2476 WSDL interface declaration as ending a conversation. This only has meaning for WSDL interfaces
2477 which are also marked conversational. The endsConversation attribute is a global attribute in the
2478 SCA namespace, with the following definition:

```
2479 <attribute name="endsConversation" type="boolean" default="false" />
```

2480

2481 The following snippet is an example of a WSDL Port Type annotated with the **requires** attribute on
2482 the portType and the **endsConversation** attribute on one of the operations:

2483

```
2484 <portType name="LoanService" sca:requires="conversational">
```

```
2485 <operation name="apply">
```

```
2486 <input message="tns:ApplicationInput" />
```

```
2487 <output message="tns:ApplicationOutput" />
```

```
2488 </operation>
```

```
2489 <operation name="cancel" sca:endsConversation="true">
```

```
2490 </operation>
```

2491

```
2492 </portType>
```

2493

2494 Binding

2495 Bindings are used by services and references. References use bindings to describe the access
2496 mechanism used to call a service (which can be a service provided by another SCA composite).
2497 Services use bindings to describe the access mechanism that clients (which can be a client from
2498 another SCA composite) have to use to call the service.

2499
2500 SCA supports the use of multiple different types of bindings. Examples include **SCA service, Web**
2501 **service, stateless session EJB, data base stored procedure, EIS service**. An SCA runtime
2502 MUST provide support for SCA service and Web service binding types. SCA provides an
2503 extensibility mechanism by which an SCA runtime can add support for additional binding types.
2504 For details on how additional binding types are defined, see the section on the Extension Model.

2505
2506 A binding is defined by a **binding element** which is a child element of a service or of a reference
2507 element in a composite. The following snippet shows the composite schema with the schema for
2508 the binding element.

```
2509  
2510 <?xml version="1.0" encoding="ASCII"?>  
2511 <!-- Bindings schema snippet -->  
2512 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
2513     targetNamespace="xs:anyURI"  
2514     name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?  
2515     constrainingType="QName"?  
2516     requires="list of xs:QName"? policySets="list of  
2517 xs:QName"?>  
2518  
2519     ...  
2520  
2521     <service name="xs:NCName" promote="xs:anyURI"  
2522         requires="list of xs:QName"? policySets="list of xs:QName"?>*  
2523         <interface ... />?  
2524         <binding uri="xs:anyURI"? name="xs:NCName"?  
2525             requires="list of xs:QName"? policySets="list of  
2526 xs:QName"? />*  
2527         <callback?>  
2528             <binding uri="xs:anyURI"? name="xs:NCName"?  
2529                 requires="list of xs:QName"?  
2530                 policySets="list of xs:QName"? />+  
2531         </callback>  
2532     </service>  
2533  
2534     ...  
2535  
2536     <reference name="xs:NCName" target="list of xs:anyURI"?>
```

Formatted: German
(Germany)

Formatted: French (France)

```

2537     promote="list of xs:anyURI"? wiredByImpl="xs:boolean"?
2538     multiplicity="0..1 or 1..1 or 0..n or 1..n"?
2539     requires="list of xs:QName"? policySets="list of xs:QName"?>*
2540 <interface ... />?
2541 <binding uri="xs:anyURI"? name="xs:NCName"?
2542     requires="list of xs:QName"? policySets="list of
2543 xs:QName"?/>*
2544 <callback?>
2545     <binding uri="xs:anyURI"? name="xs:NCName"?
2546     requires="list of xs:QName"?
2547     policySets="list of xs:QName"?/>+
2548 </callback>
2549 </reference>
2550
2551 ...
2552
2553 </composite>
2554

```

2555 The element name of the binding element is architected; it is in itself a qualified name. The first
2556 qualifier is always named "binding", and the second qualifier names the respective binding-type
2557 (e.g. binding.composite, binding.ws, binding.ejb, binding.eis).

2558

2559 A binding element has the following attributes:

- 2560 • **uri (optional)** - has the following semantic.
 - 2561 ○ For a binding of a **reference** the URI attribute defines the target URI of the
 - 2562 reference (either the component/service for a wire to an endpoint within the SCA
 - 2563 domain or the accessible address of some endpoint outside the SCA domain). It is
 - 2564 optional for references defined in composites used as component implementations,
 - 2565 but required for references defined in composites contributed to SCA domains. The
 - 2566 URI attribute of a reference of a composite can be reconfigured by a component in
 - 2567 a containing composite using the composite as an implementation. Some binding
 - 2568 types may require that the address of the target service uses more than a simple
 - 2569 URI (such as a WS-Addressing endpoint reference). In those cases, the binding
 - 2570 type will define the additional attributes or sub-elements that are necessary to
 - 2571 identify the service.
 - 2572 ○ For a binding of a **service** the URI attribute defines the URI relative to the
 - 2573 component which contributes the service to the SCA domain. The default value for
 - 2574 the URI is the the value of the name attribute of the binding.
- 2575 • **name (optional)** – a name for the binding instance (an NCName). The name attribute
- 2576 allows distinction between multiple binding elements on a single service or reference. The
- 2577 default value of the name attribute is the service or reference name. When a service or
- 2578 reference has multiple bindings, only one can have the default value; all others must have
- 2579 a value specified that is unique within the service or reference. The name also permits the
- 2580 binding instance to be referenced from elsewhere – particularly useful for some types of
- 2581 binding, which can be declared in a definitions document as a template and referenced
- 2582 from other binding instances, simplifying the definition of more complex binding instances
- 2583 (see [the JMS Binding specification \[11\]](#) for examples of this referencing).
- 2584 • **requires (optional)** - a list of policy intents. See the [Policy Framework specification \[10\]](#)
- 2585 for a description of this attribute.

2586 • **policySets (optional)** – a list of policy sets. See the [Policy Framework specification \[10\]](#)
2587 for a description of this attribute.

2588 When multiple bindings exist for an service, it means that the service is available by any of the
2589 specified bindings. The technique that the SCA runtime uses to choose among available bindings
2590 is left to the implementation and it may include additional (nonstandard) configuration. Whatever
2591 technique is used SHOULD be documented.

2592 Services and References can always have their bindings overridden at the SCA domain level,
2593 unless restricted by Intents applied to them.

2594 The following sections describe the SCA and Web service binding type in detail.

2595

2596 8.5 Messages containing Data not defined in the Service Interface

2597

2598 It is possible for a message to include information that is not defined in the interface used to
2599 define the service, for instance information may be contained in SOAP headers or as MIME
2600 attachments.

2601 Implementation types MAY make this information available to component implementations in their
2602 execution context. These implementation types must indicate how this information is accessed
2603 and in what form they are presented.

2604

2605 8.6 Form of the URI of a Deployed Binding

2606

2607 8.6.1 Constructing Hierarchical URIs

2608 Bindings that use hierarchical URI schemes construct the effective URI with a combination of the
2609 following pieces:

2610 Base System URI for a scheme / Component URI / Service Binding URI

2611

2612 Each of these components deserves addition definition:

2613 **Base Domain URI for a scheme.** An SCA domain should define a base URI for each hierarchical
2614 URI scheme on which it intends to provide services.

2615 For example: the HTTP and HTTPS schemes would each have their own base URI defined for the
2616 domain. An example of a scheme that is not hierarchical, and therefore will have no base URI is
2617 the "jms:" scheme.

2618 **Component URI.** The component URI above is for a component that is deployed in the SCA
2619 Domain. The URI of a component defaults to the name of the component, which is used as a
2620 relative URI. The component may have a specified URI value. The specified URI value may be an
2621 absolute URI in which case it becomes the Base URI for all the services belonging to the
2622 component. If the specified URI value is a relative URI, it is used as the Component URI value
2623 above.

2624 **Service Binding URI.** The Service Binding URI is the relative URI specified in the "uri" attribute
2625 of a binding element of the service. The default value of the attribute is value of the binding's
2626 name attribute treated as a relative URI. If multiple bindings for a single service use the same
2627 scheme (e.g. HTTP), then only one of the bindings may depend on the default value for the uri
2628 attribute, i.e. only one may use the default binding name. The service binding URI may also be
2629 absolute, in which case the absolute URI fully specifies the full URI of the service. Some
2630 deployment environments may not support the use of absolute URIs in service bindings.

2631 Services deployed into the Domain (as opposed to services of components) have a URI that does
2632 not include a component name, i.e.:

2633 Base Domain URI for a scheme / Service Binding URI

2634 The name of the containing composite does not contribute to the URI of any service.

2635 For example, a service where the Base URI is "http://acme.com", the component is named
2636 "stocksComponent" and the service binding name is "getQuote", the URI would look like this:

2637 http://acme.com/stocksComponent/getQuote

2638 Allowing a binding's relative URI to be specified that differs from the name of the service allows
2639 the URI hierarchy of services to be designed independently of the organization of the domain.

2640 It is good practice to design the URI hierarchy to be independent of the domain organization, but
2641 there may be times when domains are initially created using the default URI hierarchy. When this
2642 is the case, the organization of the domain can be changed, while maintaining the form of the URI
2643 hierarchy, by giving appropriate values to the *uri* attribute of select elements. Here is an example
2644 of a change that can be made to the organization while maintaining the existing URIs:

2645 To move a subset of the services out of one component (say "foo") to a new component (say
2646 "bar"), the new component should have bindings for the moved services specify a URI
2647 "../foo/MovedService"..

2648 The URI attribute may also be used in order to create shorter URIs for some endpoints, where the
2649 component name may not be present in the URI at all. For example, if a binding has a *uri*
2650 attribute of "../myService" the component name will not be present in the URI.

2651 8.6.2 Non-hierarchical URIs

2652 Bindings that use non-hierarchical URI schemes (such as jms: or mailto:) may optionally make
2653 use of the "uri" attribute, which is the complete representation of the URI for that service
2654 binding. Where the binding does not use the "uri" attribute, the binding must offer a different
2655 mechanism for specifying the service address.

2656 8.6.3 Determining the URI scheme of a deployed binding

2657 One of the things that needs to be determined when building the effective URI of a deployed
2658 binding (i.e. endpoint) is the URI scheme. The process of determining the endpoint URI scheme is
2659 binding type specific.

2660 If the binding type supports a single protocol then there is only one URI scheme associated with it.
2661 In this case, that URI scheme is used.

2662 If the binding type supports multiple protocols, the binding type implementation determines the
2663 URI scheme by introspecting the binding configuration, which may include the policy sets
2664 associated with the binding.

2665 A good example of a binding type that supports multiple protocols is binding.ws, which can be
2666 configured by referencing either an "abstract" WSDL element (i.e. portType or interface) or a
2667 "concrete" WSDL element (i.e. binding, port or endpoint). When the binding references a PortType
2668 or Interface, the protocol and therefore the URI scheme is derived from the intents/policy sets
2669 attached to the binding. When the binding references a "concrete" WSDL element, there are two
2670 cases:

- 2671 1) The referenced WSDL binding element uniquely identifies a URI scheme. This is the most
2672 common case. In this case, the URI scheme is given by the protocol/transport specified in the
2673 WSDL binding element.
- 2674 2) The referenced WSDL binding element doesn't uniquely identify a URI scheme. For example,
2675 when HTTP is specified in the @transport attribute of the SOAP binding element, both "http"
2676 and "https" could be used as valid URI schemes. In this case, the URI scheme is determined
2677 by looking at the policy sets attached to the binding.

2678 It's worth noting that an intent supported by a binding type may completely change the behavior
2679 of the binding. For example, when the intent "confidentiality/transport" is required by an HTTP
2680 binding, SSL is turned on. This basically changes the URI scheme of the binding from "http" to
2681 "https".

2682

2683 8.7 SCA Binding

2684 The SCA binding element is defined by the following schema.

2685

```
2686 <binding.sca />
```

2687

2688 The SCA binding can be used for service interactions between references and services contained
2689 within the SCA domain. The way in which this binding type is implemented is not defined by the
2690 SCA specification and it can be implemented in different ways by different SCA runtimes. The only
2691 requirement is that the required qualities of service must be implemented for the SCA binding
2692 type. The SCA binding type is **not** intended to be an interoperable binding type. For
2693 interoperability, an interoperable binding type such as the Web service binding should be used.

2694 A service definition with no binding element specified uses the SCA binding.

2695 <binding.sca/> would only have to be specified in override cases, or when you specify a
2696 set of bindings on a service definition and the SCA binding should be one of them.

2697 If a reference does not have a binding, then the binding used can be any of the bindings
2698 specified by the service provider, as long as the intents required by the reference and
2699 the service are all respected.

2700 If the interface of the service or reference is local, then the local variant of the SCA
2701 binding will be used. If the interface of the service or reference is remotable, then either
2702 the local or remote variant of the SCA binding will be used depending on whether source
2703 and target are co-located or not.

2704 If a reference specifies an URI via its uri attribute, then this provides the default wire to a service
2705 provided by another domain level component. The value of the URI has to be as follows:

- 2706 • <domain-component-name>/<service-name>

2707

2708 8.7.1 Example SCA Binding

2709 The following snippet shows the MyValueComposite.composite file for the MyValueComposite
2710 containing the service element for the MyValueService and a reference element for the
2711 StockQuoteService. Both the service and the reference use an SCA binding. The target for the
2712 reference is left undefined in this binding and would have to be supplied by the composite in which
2713 this composite is used.

2714

```
2715 <?xml version="1.0" encoding="ASCII"?>
```

```
2716 <!-- Binding SCA example -->
```

```
2717 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
```

```
2718     targetNamespace="http://foo.com"
```

```
2719     name="MyValueComposite" >
```

2720

```
2721     <service name="MyValueService" promote="MyValueComponent">
```

```
2722         <interface.java interface="services.myvalue.MyValueService"/>
```

```
2723         <binding.sca/>
2724     ...
2725 </service>
2726
2727     ...
2728
2729     <reference name="StockQuoteService"
2730 promote="MyValueComponent/StockQuoteReference">
2731         <interface.java
2732 interface="services.stockquote.StockQuoteService"/>
2733         <binding.sca/>
2734     </reference>
2735
2736 </composite>
2737
```

2738 **8.8 Web Service Binding**

2739 SCA defines a Web services binding. This is described in [a separate specification document \[9\]](#).

2740

2741 **8.9 JMS Binding**

2742 SCA defines a JMS binding. This is described in [a separate specification document \[11\]](#).

2743
2744
2745
2746
2747
2748
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2770
2771
2772
2773
2774

9 SCA Definitions

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

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

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

    <sca:intent/*>
    <sca:policySet/*>
    <sca:binding/*>
    <sca:bindingType/*>
    <sca:implementationType/*>
</definitions>
```

Formatted: English (U.S.)

The definitions element has the following attribute:

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

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

2775

10 Extension Model

2776

2777 The assembly model can be extended with support for new interface types, implementation types
2778 and binding types. The extension model is based on XML schema substitution groups. There are
2779 three XML Schema substitution group heads defined in the SCA namespace: **interface**,
2780 **implementation** and **binding**, for interface types, implementation types and binding types,
2781 respectively.

2782 The SCA Client and Implementation specifications and the SCA Bindings specifications (see [1])
2783 use these XML Schema substitution groups to define some basic types of interfaces,
2784 implementations and bindings, but other types can be defined as required, where support for
2785 these extra ones is available from the runtime. The interface type elements, implementation type
2786 elements, and binding type elements defined by the SCA specifications (see [1]) are all part of the
2787 SCA namespace ("http://docs.oasis-open.org/ns/opencsa/sca/200712"), as indicated in their
2788 respective schemas. New interface types, implementation types and binding types that are defined
2789 using this extensibility model, which are not part of these SCA specifications must be defined in
2790 namespaces other than the SCA namespace.

2791 The "." notation is used in naming elements defined by the SCA specifications (e.g.
2792 <implementation.java ... />, <interface.wsdl ... />, <binding.ws ... />), not as a parallel
2793 extensibility approach but as a naming convention that improves usability of the SCA assembly
2794 language.

2795

2796 **Note:** How to contribute SCA model extensions and their runtime function to an SCA runtime will
2797 be defined by a future version of the specification.

2798

10.1 Defining an Interface Type

2800 The following snippet shows the base definition for the **interface** element and **Interface** type
2801 contained in **sca-core.xsd**; see appendix for complete schema.

2802

```
2803 <?xml version="1.0" encoding="UTF-8"?>  
2804 <!-- (c) Copyright SCA Collaboration 2006 -->  
2805 <schema xmlns="http://www.w3.org/2001/XMLSchema"  
2806         targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
2807         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
2808         elementFormDefault="qualified">
```

Formatted: English (U.S.)

2809

2810 ...

2811

```
2812     <element name="interface" type="sca:Interface" abstract="true"/>
```

```
2813     <complexType name="Interface"/>
```

2814

2815 ...

2816

```
2817 </schema>
```

2818 In the following snippet we show how the base definition is extended to support Java interfaces.
2819 The snippet shows the definition of the **interface.java** element and the **JavaInterface** type
2820 contained in **sca-interface-java.xsd**.

```
2821  
2822 <?xml version="1.0" encoding="UTF-8"?>  
2823 <schema xmlns="http://www.w3.org/2001/XMLSchema"  
2824         targetNamespace="http://docs.oasis-  
2825 open.org/ns/opencsa/sca/200712"  
2826         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712">  
2827  
2828     <element name="interface.java" type="sca:JavaInterface"  
2829         substitutionGroup="sca:interface" />  
2830     <complexType name="JavaInterface">  
2831         <complexContent>  
2832             <extension base="sca:Interface">  
2833                 <attribute name="interface" type="NCName"  
2834 use="required" />  
2835             </extension>  
2836         </complexContent>  
2837     </complexType>  
2838 </schema>
```

Formatted: English (U.S.)

2839 In the following snippet we show an example of how the base definition can be extended by other
2840 specifications to support a new interface not defined in the SCA specifications. The snippet shows
2841 the definition of the **my-interface-extension** element and the **my-interface-extension-type**
2842 type.

```
2843 <?xml version="1.0" encoding="UTF-8"?>  
2844 <schema xmlns="http://www.w3.org/2001/XMLSchema"  
2845         targetNamespace="http://www.example.org/myextension"  
2846         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
2847         xmlns:tns="http://www.example.org/myextension">  
2848  
2849     <element name="my-interface-extension" type="tns:my-interface-  
2850 extension-type"  
2851         substitutionGroup="sca:interface" />  
2852     <complexType name="my-interface-extension-type">  
2853         <complexContent>  
2854             <extension base="sca:Interface">  
2855                 ...  
2856             </extension>  
2857         </complexContent>  
2858     </complexType>  
2859 </schema>  
2860
```

2861 10.2 Defining an Implementation Type

2862 The following snippet shows the base definition for the *implementation* element and
2863 *Implementation* type contained in *sca-core.xsd*; see appendix for complete schema.

```
2864  
2865 <?xml version="1.0" encoding="UTF-8"?>  
2866 <!-- (c) Copyright SCA Collaboration 2006 -->  
2867 <schema xmlns="http://www.w3.org/2001/XMLSchema"  
2868         targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
2869         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
2870         elementFormDefault="qualified">  
2871  
2872     ...  
2873  
2874     <element name="implementation" type="sca:Implementation"  
2875     abstract="true"/>  
2876     <complexType name="Implementation"/>  
2877  
2878     ...  
2879 </schema>
```

Formatted: English (U.S.)

2881
2882 In the following snippet we show how the base definition is extended to support Java
2883 implementation. The snippet shows the definition of the *implementation.java* element and the
2884 *JavaImplementation* type contained in *sca-implementation-java.xsd*.

```
2885  
2886 <?xml version="1.0" encoding="UTF-8"?>  
2887 <schema xmlns="http://www.w3.org/2001/XMLSchema"  
2888         targetNamespace="http://docs.oasis-  
2889 open.org/ns/opencsa/sca/200712"  
2890         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712">  
2891  
2892     <element name="implementation.java" type="sca:JavaImplementation"  
2893             substitutionGroup="sca:implementation"/>  
2894     <complexType name="JavaImplementation">  
2895         <complexContent>  
2896             <extension base="sca:Implementation">  
2897                 <attribute name="class" type="NCName"  
2898 use="required"/>  
2899             </extension>  
2900         </complexContent>  
2901     </complexType>  
2902 </schema>
```

2903 In the following snippet we show an example of how the base definition can be extended by other
2904 specifications to support a new implementation type not defined in the SCA specifications. The

2905 snippet shows the definition of the **my-impl-extension** element and the **my-impl-extension-**
2906 **type** type.

```
2907 <?xml version="1.0" encoding="UTF-8"?>
2908 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2909         targetNamespace="http://www.example.org/myextension"
2910         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2911         xmlns:tns="http://www.example.org/myextension">
2912
2913     <element name="my-impl-extension" type="tns:my-impl-extension-type"
2914             substitutionGroup="sca:implementation"/>
2915     <complexType name="my-impl-extension-type">
2916         <complexContent>
2917             <extension base="sca:Implementation">
2918                 ▲·····
2919             </extension>
2920         </complexContent>
2921     </complexType>
2922 </schema>
```

Formatted: French (France)

2924 In addition to the definition for the new implementation instance element, there needs to be an
2925 associated implementationType element which provides metadata about the new implementation
2926 type. The pseudo schema for the implementationType element is shown in the following snippet:

```
2927 <implementationType type="xs:QName"
2928                   alwaysProvides="list of intent xs:QName"
2929                   mayProvide="list of intent xs:QName"/>
```

2931 The implementation type has the following attributes:

- 2932 • **type (required)** – the type of the implementation to which this implementationType
2933 element applies. This is intended to be the QName of the implementation element for the
2934 implementation type, such as "sca:implementation.java"
- 2935 • **alwaysProvides (optional)** – a set of intents which the implementation type always
2936 provides. See [the Policy Framework specification \[10\]](#) for details.
- 2937 • **mayProvide (optional)** – a set of intents which the implementation type may provide.
2938 See [the Policy Framework specification \[10\]](#) for details.

2939

2940 10.3 Defining a Binding Type

2941 The following snippet shows the base definition for the **binding** element and **Binding** type
2942 contained in **sca-core.xsd**; see appendix for complete schema.

```
2943
2944 <?xml version="1.0" encoding="UTF-8"?>
2945 <!-- binding type schema snippet -->
2946 <!-- (c) Copyright SCA Collaboration 2006, 2007 -->
2947 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2948         targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712">
```

```

2949         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2950         elementFormDefault="qualified">
2951     ...
2952
2953
2954         <element name="binding" type="sca:Binding" abstract="true"/>
2955         <complexType name="Binding">
2956             <attribute name="uri" type="anyURI" use="optional"/>
2957             <attribute name="name" type="NCName" use="optional"/>
2958             <attribute name="requires" type="sca:listOfQNames"
2959 use="optional"/>
2960             <attribute name="policySets" type="sca:listOfQNames"
2961 use="optional"/>
2962         </complexType>
2963
2964     ...
2965
2966 </schema>

```

2967 In the following snippet we show how the base definition is extended to support Web service
2968 binding. The snippet shows the definition of the **binding.ws** element and the
2969 **WebServiceBinding** type contained in **sca-binding-webservice.xsd**.

```

2970
2971 <?xml version="1.0" encoding="UTF-8"?>
2972 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2973         targetNamespace="http://docs.oasis-
2974 open.org/ns/opencsa/sca/200712"
2975         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712">
2976
2977     <element name="binding.ws" type="sca:WebServiceBinding"
2978         substitutionGroup="sca:binding"/>
2979     <complexType name="WebServiceBinding">
2980         <complexContent>
2981             <extension base="sca:Binding">
2982                 <attribute name="port" type="anyURI" use="required"/>
2983             </extension>
2984         </complexContent>
2985     </complexType>
2986 </schema>

```

Formatted: English (U.S.)

2987 In the following snippet we show an example of how the base definition can be extended by other
2988 specifications to support a new binding not defined in the SCA specifications. The snippet shows
2989 the definition of the **my-binding-extension** element and the **my-binding-extension-type** type.

```

2990 <?xml version="1.0" encoding="UTF-8"?>
2991 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2992         targetNamespace="http://www.example.org/myextension"
2993         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712">

```

```

2994         xmlns:tns="http://www.example.org/myextension">
2995
2996         <element name="my-binding-extension" type="tns:my-binding-extension-
2997 type"
2998             substitutionGroup="sca:binding"/>
2999         <complexType name="my-binding-extension-type">
3000             <complexContent>
3001                 <extension base="sca:Binding">
3002                     ...
3003                 </extension>
3004             </complexContent>
3005         </complexType>
3006 </schema>
3007

```

Formatted: English (U.S.)

3008 In addition to the definition for the new binding instance element, there needs to be an associated
3009 bindingType element which provides metadata about the new binding type. The pseudo schema
3010 for the bindingType element is shown in the following snippet:

```

3011 <bindingType type="xs:QName"
3012             alwaysProvides="list of intent QNames"?
3013             mayProvide = "list of intent QNames"?/>
3014

```

3015 The binding type has the following attributes:

- 3016 • **type (required)** – the type of the binding to which this bindingType element applies.
3017 This is intended to be the QName of the binding element for the binding type, such as
3018 "sca:binding.ws"
- 3019 • **alwaysProvides (optional)** – a set of intents which the binding type always provides.
3020 See [the Policy Framework specification \[10\]](#) for details.
- 3021 • **mayProvide (optional)** – a set of intents which the binding type may provide. See [the](#)
3022 [Policy Framework specification \[10\]](#) for details.

3023 11 Packaging and Deployment

3024 11.1 Domains

3025 An **SCA Domain** represents a complete runtime configuration, potentially distributed over a series
3026 of interconnected runtime nodes.

3027 A single SCA domain defines the boundary of visibility for all SCA mechanisms. For example, SCA
3028 wires can only be used to connect components within a single SCA domain. Connections to
3029 services outside the domain must use binding specific mechanisms for addressing services (such
3030 as WSDL endpoint URIs). Also, SCA mechanisms such as intents and policySets can only be used
3031 in the context of a single domain. In general, external clients of a service that is developed and
3032 deployed using SCA should not be able to tell that SCA was used to implement the service – it is
3033 an implementation detail.

3034 The size and configuration of an SCA Domain is not constrained by the SCA Assembly specification
3035 and is expected to be highly variable. An SCA Domain typically represents an area of business
3036 functionality controlled by a single organization. For example, an SCA Domain may be the whole
3037 of a business, or it may be a department within a business.

3038 As an example, for the accounts department in a business, the SCA Domain might cover all
3039 finance-related functions, and it might contain a series of composites dealing with specific areas of
3040 accounting, with one for Customer accounts and another dealing with Accounts Payable.

3041 An SCA domain has the following:

- 3042 • A virtual domain-level composite whose components are deployed and running
- 3043 • A set of *installed contributions* that contain implementations, interfaces and other artifacts
3044 necessary to execute components
- 3045 • A set of logical services for manipulating the set of contributions and the virtual domain-
3046 level composite.

3047 The information associated with an SCA domain can be stored in many ways, including but not
3048 limited to a specific filesystem structure or a repository.

3049 11.2 Contributions

3050 An SCA domain may require a large number of different artifacts in order to work. These artifacts
3051 include artifacts defined by SCA and other artifacts such as object code files and interface
3052 definition files. The SCA-defined artifact types are all XML documents. The root elements of the
3053 different SCA definition documents are: `composite`, `componentType`, `constrainingType` and
3054 `definitions`. XML artifacts that are not defined by SCA but which may be needed by an SCA
3055 domain include XML Schema documents, WSDL documents, and BPEL documents. SCA
3056 constructs, like other XML-defined constructs, use XML qualified names for their identity (i.e.
3057 namespace + local name).

3058 Non-XML artifacts are also required within an SCA domain. The most obvious examples of such
3059 non-XML artifacts are Java, C++ and other programming language files necessary for component
3060 implementations. Since SCA is extensible, other XML and non-XML artifacts may also be required.

3061 SCA defines an interoperable packaging format for contributions (ZIP), as specified below. This
3062 format is not the only packaging format that an SCA runtime can use. SCA allows many different
3063 packaging formats, but requires that the ZIP format be supported. When using the ZIP format for
3064 deploying a contribution, this specification does not specify whether that format is retained after
3065 deployment. For example, a Java EE based SCA runtime may convert the ZIP package to an EAR
3066 package. SCA expects certain characteristics of any packaging:

- 3067 • It must be possible to present the artifacts of the packaging to SCA as a hierarchy of
3068 resources based off of a single root

- 3069
- A directory resource should exist at the root of the hierarchy named META-INF
- 3070
- A document should exist directly under the META-INF directory named sca-
- 3071
- 3072
- 3073
- 3074
- 3075
- 3076
- 3077
- 3078
- 3079
- 3080
- 3081
- 3082

The same document also optionally lists namespaces of constructs that are defined within the contribution and which may be used by other contributions

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

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

3083 To illustrate that a variety of packaging formats can be used with SCA, the following are examples

3084 of formats that might be used to package SCA artifacts and metadata (as well as other artifacts)

3085 as a contribution:

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

3090 Contributions do not contain other contributions. If the packaging format is a JAR file that

3091 contains other JAR files (or any similar nesting of other technologies), the internal files are not

3092 treated as separate SCA contributions. It is up to the implementation to determine whether the

3093 internal JAR file should be represented as a single artifact in the contribution hierarchy or whether

3094 all of the contents should be represented as separate artifacts.

3095 A goal of SCA's approach to deployment is that the contents of a contribution should not need to

3096 be modified in order to install and use the contents of the contribution in a domain.

3097

3098 11.2.1 SCA Artifact Resolution

3099 Contributions can be self-contained, in that all of the artifacts necessary to run the contents of the

3100 contribution are found within the contribution itself. However, it can also be the case that the

3101 contents of the contribution make one or many references to artifacts that are not contained

3102 within the contribution. These references may be to SCA artifacts such as composites or they may

3103 be to other artifacts such as WSDL files, XSD files or to code artifacts such as Java class files and

3104 BPEL process files.

Deleted: may

Deleted: may

Deleted: script

Deleted: s

Deleted: may

3105 A contribution can use some artifact-related or packaging-related means to resolve artifact

3106 references. Examples of such mechanisms include:

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

3110 Where present, these mechanisms MUST be used by the SCA runtime to resolve artifact

3111 dependencies.

Deleted: must

3112 SCA also provides an artifact resolution mechanism. The SCA artifact resolution mechanism is

3113 used either where no other mechanisms are available, for example in cases where the mechanisms

3114 used by the various contributions in the same SCA Domain are different. An example of the latter

3115 is where an OSGi Bundle is used for one contribution but where a second contribution used by the

3116 first one is not implemented using OSGi - eg the second contribution relates to a mainframe

3117 COBOL service whose interfaces are declared using a WSDL which must be accessed by the first

3118 contribution.

Deleted: s are

Deleted: , or

Deleted: case

Deleted: is

3119 The SCA artifact resolution is likely to be most useful for SCA domains containing heterogeneous
3120 mixtures of contribution, where artifact-related or packaging-related mechanisms are unlikely to
3121 work across different kinds of contributions.

3122 SCA artifact resolution works on the principle that a contribution which needs to use artifacts
3123 defined elsewhere expresses these dependencies using **import** statements in metadata belonging
3124 to the contribution. A contribution controls which artifacts it makes available to other
3125 contributions through **export** statements in metadata attached to the contribution. [SCA artifact
3126 resolution is a general mechanism that can be extended for the handling of specific types of
3127 artifact. The general mechanism that is described in the following paragraphs is mainly intended
3128 for the handling of XML artifacts. Other types of artifacts, for example Java classes, use an
3129 extended version of artifact resolution that is specialized to their nature \(eg. instead of
3130 "namespaces", Java uses "packages"\). Descriptions of these more specialized forms of artifact
3131 resolution are contained in the SCA specifications that deal with those artifact types.](#)

3132 [Import and export statements for XML artifacts work at the level of namespaces - so that an
3133 import statement declares that artifacts from a specified namespace are found in other
3134 contributions, while an export statement makes all the artifacts from a specified namespace
3135 available to other contributions.](#)

3136 [An import declaration can simply specify the namespace to import. In this case, the locations
3137 which are searched for artifacts in that namespace are the contribution\(s\) in the Domain which
3138 have export declarations for the same namespace, if any. Alternatively an import declaration can
3139 specify a location from which artifacts for the namespace are obtained, in which case, that specific
3140 location is searched. There can be multiple import declarations for a given namespace. Where
3141 multiple declarations are made for the same namespace, all the locations specified are searched
3142 \(without any implied ordering\).](#)

3143 [For an XML namespace, artifacts may be declared in multiple locations - for example a given
3144 namespace may have a WSDL declared in one contribution and have an XSD defining XML data
3145 types in a second contribution.](#)

3146 [If the same artifact is declared in multiple locations, this is not an error. It is implementation
3147 dependent which version of the artifact is selected in this case.](#)

3148 [When a contribution contains a reference to an artifact from a namespace that is declared in an
3149 import statement of the contribution, if the SCA artifact resolution mechanism is used to resolve
3150 the artifact, the SCA runtime MUST resolve artifacts in the following order:](#)

- 3151 1. [from the locations identified by the import statement\(s\) for the namespace. Locations
3152 MUST NOT be searched recursively in order to locate artifacts \(ie only a one-level search is
3153 performed\).](#)
- 3154 2. [from the contents of the contribution itself.](#)

3155 [When a contribution uses an artifact contained in another contribution through SCA artifact
3156 resolution, if that artifact itself has dependencies on other artifacts, the SCA runtime MUST resolve
3157 these dependencies in the context of the contribution containing the artifact, not in the context of
3158 the original contribution.](#)

3159 [For example:](#)

- 3160 • [a first contribution "C1" references an artifact "A1" in the namespace "n1" and imports the
3161 "n1" namespace from a second contribution "C2".](#)
- 3162 • [in contribution "C2" the artifact "A1" in the "n1" namespace references an artifact "A2"
3163 also in the "n1" namespace", which is resolved through an import of the "n1" namespace
3164 in "C2" which specifies the location "C3".](#)
- 3165 • [The "A2" artifact is contained within the third contribution "C3" from which it is resolved
3166 by the contribution "C2". The "C3" contribution is never used to resolve artifacts directly
3167 for the "C1" contribution, since "C3" is not declared as an import location for "C1".](#)

3169 [For example, if for a contribution "C1", an import is used to resolve a composite "X1" contained in
3170 contribution "C2", and composite "X1" contains references to other artifacts such as WSDL files or](#)

Deleted: When a contribution uses one or more artifacts from another contribution through the import mechanism, there is a need to be clear about how artifacts are resolved.

Deleted: access

3171 XSDs, those references in "X1" are resolved in the context of contribution "C2" and not in the
 3172 context of contribution "C1".
 3173 The SCA runtime MUST ignore local definitions of an artifact if the artifact is found through
 3174 resolving an import statement.
 3175 The SCA runtime MUST raise an error if an artifact cannot be resolved by the precedence order
 3176 above.
 3177

Deleted: it is important to ensure that

Deleted: X1

Deleted: which uses

Deleted: X

Deleted: When a contribution is using the SCA artifact resolution mechanism, the SCA runtime MUST resolve artifacts in the following order:¶
 <#>from the contribution's direct dependencies by resolving its sca import statements. Dependencies MUST NOT be searched recursively in order to locate artifacts.¶
 <#>from the contents of the contribution itself.¶
 For example, a first contribution imports a namespace n1 from a second contribution. The n1 namespace definition references artifacts x1 that are imported by the second contribution. The x1 artifacts are contained within a third contribution from which they are resolved by the second contribution. The third contribution is never used to provide artifacts for the first contribution.¶

3178 11.2.2 SCA Contribution Metadata Document

3179 The contribution optionally contains a document that declares runnable composites, exported
 3180 definitions and imported definitions. The document is found at the path of META-INF/sca-
 3181 contribution.xml relative to the root of the contribution. Frequently some SCA metadata may
 3182 need to be specified by hand while other metadata is generated by tools (such as the <import>
 3183 elements described below). To accommodate this, it is also possible to have an identically
 3184 structured document at META-INF/sca-contribution-generated.xml. If this document exists (or is
 3185 generated on an as-needed basis), it will be merged into the contents of sca-contribution.xml,
 3186 with the entries in sca-contribution.xml taking priority if there are any conflicting declarations.
 3187

3188 The format of the document is:

```
3189 <?xml version="1.0" encoding="ASCII"?>
3190 <!-- sca-contribution pseudo-schema -->
3191 <contribution xmlns=http://docs.oasis-open.org/ns/opencsa/sca/200712>
3192
3193     <deployable composite="xs:QName"/>*
3194     <import namespace="xs:String" location="xs:AnyURI"?/>*
3195     <export namespace="xs:String"/>*
3196
3197 </contribution>
```

3199 **deployable element:** Identifies a composite which is a composite within the contribution that is a
 3200 composite intended for potential inclusion into the virtual domain-level composite. Other
 3201 composites in the contribution are not intended for inclusion but only for use by other composites.
 3202 New composites can be created for a contribution after it is installed, by using the [add Deployment](#)
 3203 [Composite](#) capability and the add To Domain Level Composite capability.

- 3204 • **composite (required)** – The QName of a composite within the contribution.

3205

3206 **Export element:** A declaration that artifacts belonging to a particular namespace are exported
 3207 and are available for use within other contributions. An export declaration in a contribution
 3208 specifies a namespace, all of whose definitions are considered to be exported. By default,
 3209 definitions are not exported.

3210 The SCA artifact export is useful for SCA domains containing heterogeneous mixtures of
 3211 contribution packagings and technologies, where artifact-related or packaging-related mechanisms
 3212 are unlikely to work across different kinds of contribution.

- 3213 • **namespace (required)** – For XML definitions, which are identified by QNames, the
 3214 namespace should be the namespace URI for the exported definitions. For XML
 3215 technologies that define multiple *symbol spaces* that can be used within one namespace
 3216 (e.g. WSDL port types are a different symbol space from WSDL bindings), all definitions
 3217 from all symbol spaces are exported.

3218 Technologies that use naming schemes other than QNames must use a different export
 3219

3220 element from the same substitution group as the the SCA <export> element. The
3221 element used identifies the technology, and may use any value for the namespace that is
3222 appropriate for that technology. For example, <export.java> can be used can be used to
3223 export java definitions, in which case the namespace should be a fully qualified package
3224 name.

3225
3226 **Import element:** Import declarations specify namespaces of definitions that are needed by the
3227 definitions and implementations within the contribution, but which are not present in the
3228 contribution. It is expected that in most cases import declarations will be generated based on
3229 introspection of the contents of the contribution. In this case, the import declarations would be
3230 found in the META-INF/ sca-contribution-generated.xml document.

3231 • **namespace (required)** – For XML definitions, which are identified by QNames, the
3232 namespace should be the namespace URI for the imported definitions. For XML
3233 technologies that define multiple *symbol spaces* that can be used within one namespace
3234 (e.g. WSDL port types are a different symbol space from WSDL bindings), all definitions
3235 from all symbol spaces are imported.

3236
3237 Technologies that use naming schemes other than QNames must use a different import
3238 element from the same substitution group as the the SCA <import> element. The
3239 element used identifies the technology, and may use any value for the namespace that is
3240 appropriate for that technology. For example, <import.java> can be used can be used to
3241 import java definitions, in which case the namespace should be a fully qualified package
3242 name.

3243 • **location (optional)** – a URI to resolve the definitions for this import. SCA makes no
3244 specific requirements for the form of this URI, nor the means by which it is resolved. It
3245 may point to another contribution (through its URI) or it may point to some location
3246 entirely outside the SCA Domain.

3247
3248 It is expected that SCA runtimes may define implementation specific ways of resolving location
3249 information for artifact resolution between contributions. These mechanisms will however usually
3250 be limited to sets of contributions of one runtime technology and one hosting environment.

3251 In order to accommodate imports of artifacts between contributions of disparate runtime
3252 technologies, it is strongly suggested that SCA runtimes honor SCA contribution URIs as location
3253 specification.

3254 SCA runtimes that support contribution URIs for cross-contribution resolution of SCA artifacts
3255 should do so similarly when used as @schemaLocation and @wsdlLocation and other artifact
3256 location specifications.

3257 The order in which the import statements are specified may play a role in this mechanism. Since
3258 definitions of one namespace can be distributed across several artifacts, multiple import
3259 declarations can be made for one namespace.
3260

3261 The location value is only a default, and dependent contributions listed in the call to
3262 installContribution should override the value if there is a conflict. However, the specific
3263 mechanism for resolving conflicts between contributions that define conflicting definitions is
3264 implementation specific.

3265
3266 If the value of the location attribute is an SCA contribution URI, then the contribution packaging
3267 may become dependent on the deployment environment. In order to avoid such a dependency,
3268 dependent contributions should be specified only when deploying or updating contributions as
3269 specified in the section 'Operations for Contributions' below.

3270 11.2.3 Contribution Packaging using ZIP

3271 SCA allows many different packaging formats that SCA runtimes can support, but SCA requires
3272 that all runtimes support the ZIP packaging format for contributions. This format allows that

3273 metadata specified by the section 'SCA Contribution Metadata Document' be present. Specifically,
3274 it may contain a top-level "META-INF" directory and a "META-INF/sca-contribution.xml" file and
3275 there may also be an optional "META-INF/sca-contribution-generated.xml" file in the package. SCA
3276 defined artifacts as well as non-SCA defined artifacts such as object files, WSDL definition, Java
3277 classes may be present anywhere in the ZIP archive,

3278 A up to date definition of the ZIP file format is published by PKWARE in [an Application Note on the](#)
3279 [.ZIP file format \[12\]](#).

3280

3281 11.3 Installed Contribution

3282 As noted in the section above, the contents of a contribution should not need to be modified in
3283 order to install and use it within a domain. An *installed contribution* is a contribution with all of
3284 the associated information necessary in order to execute *deployable composites* within the
3285 contribution.

3286 An installed contribution is made up of the following things:

- 3287 • Contribution Packaging – the contribution that will be used as the starting point for
3288 resolving all references
- 3289 • Contribution base URI
- 3290 • Dependent contributions: a set of snapshots of other contributions that are used to resolve
3291 the import statements from the root composite and from other dependent contributions
 - 3292 ○ Dependent contributions may or may not be shared with other installed
3293 contributions.
 - 3294 ○ When the snapshot of any contribution is taken is implementation defined, ranging
3295 from the time the contribution is installed to the time of execution
- 3296 • Deployment-time composites.
3297 These are composites that are added into an installed contribution after it has been
3298 deployed. This makes it possible to provide final configuration and access to
3299 implementations within a contribution without having to modify the contribution. These
3300 are optional, as composites that already exist within the contribution may also be used for
3301 deployment.

3302

3303 Installed contributions provide a context in which to resolve qualified names (e.g. QNames in XML,
3304 fully qualified class names in Java).

3305 If multiple dependent contributions have exported definitions with conflicting qualified names, the
3306 algorithm used to determine the qualified name to use is implementation dependent.
3307 Implementations of SCA may also generate an error if there are conflicting names.

3308

3309 11.3.1 Installed Artifact URIs

3310 When a contribution is installed, all artifacts within the contribution are assigned URIs, which are
3311 constructed by starting with the base URI of the contribution and adding the relative URI of each
3312 artifact (recalling that SCA requires that any packaging format be able to offer up its artifacts in a
3313 single hierarchy).

3314

3315 11.4 Operations for Contributions

3316 SCA Domains provide the following conceptual functionality associated with contributions
3317 (meaning the function may not be represented as addressable services and also meaning that

3318 equivalent functionality may be provided in other ways). The functionality is optional meaning that
3319 some SCA runtimes may choose not to provide that functionality in any way:

3320 **11.4.1 install Contribution & update Contribution**

3321 Creates or updates an installed contribution with a supplied root contribution, and installed at a
3322 supplied base URI. A supplied dependent contribution list specifies the contributions that should
3323 be used to resolve the dependencies of the root contribution and other dependent contributions.
3324 These override any dependent contributions explicitly listed via the location attribute in the import
3325 statements of the contribution.
3326

3327 SCA follows the simplifying assumption that the use of a contribution for resolving anything also
3328 means that all other exported artifacts can be used from that contribution. Because of this, the
3329 dependent contribution list is just a list of installed contribution URIs. There is no need to specify
3330 what is being used from each one.
3331

3332 Each dependent contribution is also an installed contribution, with its own dependent
3333 contributions. By default these dependent contributions of the dependent contributions (which we
3334 will call *indirect dependent contributions*) are included as dependent contributions of the installed
3335 contribution. However, if a contribution in the dependent contribution list exports any conflicting
3336 definitions with an indirect dependent contribution, then the indirect dependent contribution is not
3337 included (i.e. the explicit list overrides the default inclusion of indirect dependent contributions).
3338 Also, if there is ever a conflict between two indirect dependent contributions, then the conflict
3339 must be resolved by an explicit entry in the dependent contribution list.

3340 Note that in many cases, the dependent contribution list can be generated. In particular, if a
3341 domain is careful to avoid creating duplicate definitions for the same qualified name, then it is
3342 easy for this list to be generated by tooling.

3343 **11.4.2 add Deployment Composite & update Deployment Composite**

3344 Adds or updates a deployment composite using a supplied composite ("composite by value" – a
3345 data structure, not an existing resource in the domain) to the contribution identified by a supplied
3346 contribution URI. The added or updated deployment composite is given a relative URI that
3347 matches the @name attribute of the composite, with a ".composite" suffix. Since all composites
3348 must run within the context of a installed contribution (any component implementations or other
3349 definitions are resolved within that contribution), this functionality makes it possible for the
3350 deployer to create a composite with final configuration and wiring decisions and add it to an
3351 installed contribution without having to modify the contents of the root contribution.

3352 Also, in some use cases, a contribution may include only implementation code (e.g. PHP scripts).
3353 It should then be possible for those to be given component names by a (possibly generated)
3354 composite that is added into the installed contribution, without having to modify the packaging.

3355 **11.4.3 remove Contribution**

3356 Removes the deployed contribution identified by a supplied contribution URI.

3357

3358 **11.5 Use of Existing (non-SCA) Mechanisms for Resolving Artifacts**

3359

3360 For certain types of artifact, there are existing and commonly used mechanisms for referencing a
3361 specific concrete location where the artifact can be resolved.

3362 Examples of these mechanisms include:

- 3363 • For WSDL files, the **@wsdlLocation** attribute is a hint that has a URI value pointing to the
3364 place holding the WSDL itself.

- 3365
- For XSDs, the **@schemaLocation** attribute is a hint which matches the namespace to a URI where the XSD is found.
- 3366

3367 **Note:** In neither of these cases is the runtime obliged to use the location hint and the URI does
3368 not have to be dereferenced.

3369 SCA permits the use of these mechanisms. Where present, these mechanisms take precedence
3370 over the SCA mechanisms. However, use of these mechanisms is discouraged because tying
3371 assemblies to addresses in this way makes the assemblies less flexible and prone to errors when
3372 changes are made to the overall SCA Domain.

3373 **Note:** If one of these mechanisms is present, but there is a failure to find the resource indicated
3374 when using the mechanism (eg the URI is incorrect or invalid, say) the SCA runtime MUST raise
3375 an error and MUST NOT attempt to use SCA resolution mechanisms as an alternative.

3376

3377 11.6 Domain-Level Composite

3378 The domain-level composite is a virtual composite, in that it is not defined by a composite
3379 definition document. Rather, it is built up and modified through operations on the domain.
3380 However, in other respects it is very much like a composite, since it contains components, wires,
3381 services and references.

3382 The abstract domain-level functionality for modifying the domain-level composite is as follows,
3383 although a runtime may supply equivalent functionality in a different form:

3384 11.6.1 add To Domain-Level Composite

3385 This functionality adds the composite identified by a supplied URI to the Domain Level Composite.
3386 The supplied composite URI must refer to a composite within a installed contribution. The
3387 composite's installed contribution determines how the composite's artifacts are resolved (directly
3388 and indirectly). The supplied composite is added to the domain composite with semantics that
3389 correspond to the domain-level composite having an <include> statement that references the
3390 supplied composite. All of the composite's components become *top-level* components and the
3391 services become externally visible services (eg. they would be present in a WSDL description of
3392 the domain).

3393 11.6.2 remove From Domain-Level Composite

3394 Removes from the Domain Level composite the elements corresponding to the composite
3395 identified by a supplied composite URI. This means that the removal of the components, wires,
3396 services and references originally added to the domain level composite by the identified
3397 composite.

3398 11.6.3 get Domain-Level Composite

3399 Returns a <composite> definition that has an <include> line for each composite that had been
3400 added to the domain level composite. It is important to note that, in dereferencing the included
3401 composites, any referenced artifacts must be resolved in terms of that installed composite.

3402 11.6.4 get QName Definition

3403 In order to make sense of the domain-level composite (as returned by get Domain-Level
3404 Composite), it must be possible to get the definitions for named artifacts in the included
3405 composites. This functionality takes the supplied URI of an installed contribution (which provides
3406 the context), a supplied qualified name of a definition to look up, and a supplied symbol space (as
3407 a QName, eg wsdl:PortType). The result is a single definition, in whatever form is appropriate for
3408 that definition type.

3409 Note that this, like all the other domain-level operations, is a conceptual operation. Its capabilities
3410 should exist in some form, but not necessarily as a service operation with exactly this signature.

3411

12 Conformance

3412

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

3413

3414

A. Pseudo Schema

3415

A.1 ComponentType

3416

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

3417

```
<!-- Component type schema snippet -->
```

3418

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

3419

```
  constrainingType="QName"? >
```

3420

3421

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

3422

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

3423

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

3424

```
    <binding uri="xs:anyURI"? name="xs:NCName"?
```

3425

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

3426

```
      policySets="list of xs:QName"?/>*
```

3427

```
    <callback?>
```

3428

```
      <binding ... />+
```

3429

```
    </callback>
```

3430

```
</service>
```

3431

3432

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

3433

```
  target="list of xs:anyURI"? autowire="xs:boolean"?
```

3434

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

3435

```
  wiredByImpl="xs:boolean"? requires="list of xs:QName"?
```

3436

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

3437

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

3438

```
  <binding uri="xs:anyURI"? name="xs:NCName"?
```

3439

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

3440

```
    policySets="list of xs:QName"?/>*
```

3441

```
  <callback?>
```

3442

```
    <binding ... />+
```

3443

```
  </callback>
```

3444

```
</reference>
```

3445

3446

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

3447

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

3448

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

3449

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

3450

```
</property>
```

3451

3452

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

3453

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

3454
3455 </componentType>
3456

3457 A.2 Composite

```
3458 <?xml version="1.0" encoding="ASCII"?>
3459 <!-- Composite schema snippet -->
3460 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3461           targetNamespace="xs:anyURI"
3462           name="xs:NCName" local="xs:boolean"?
3463           autowire="xs:boolean"? constrainingType="QName"?
3464           requires="list of xs:QName"? policySets="list of
3465 xs:QName"?>
3466
3467   <include name="xs:QName"/>*
3468
3469   <service name="xs:NCName" promote="xs:anyURI"
3470           requires="list of xs:QName"? policySets="list of xs:QName"?>*
3471   <interface ... />?
3472   <binding uri="xs:anyURI"? name="xs:NCName"?
3473           requires="list of xs:QName"? policySets="list of
3474 xs:QName"?/>*
3475   <callback?>
3476     <binding uri="xs:anyURI"? name="xs:NCName"?
3477           requires="list of xs:QName"?
3478           policySets="list of xs:QName"?/>+
3479   </callback>
3480 </service>
3481
3482 <reference name="xs:NCName" target="list of xs:anyURI"?
3483           promote="list of xs:anyURI" wiredByImpl="xs:boolean"?
3484           multiplicity="0..1 or 1..1 or 0..n or 1..n"?
3485           requires="list of xs:QName"? policySets="list of xs:QName"?>*
3486 <interface ... />?
3487 <binding uri="xs:anyURI"? name="xs:NCName"?
3488           requires="list of xs:QName"? policySets="list of
3489 xs:QName"?/>*
3490 <callback?>
3491   <binding uri="xs:anyURI"? name="xs:NCName"?
3492           requires="list of xs:QName"?
3493           policySets="list of xs:QName"?/>+
3494 </callback>
3495 </reference>
3496
```

```

3497     <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
3498         many="xs:boolean"? mustSupply="xs:boolean"?>*
3499         default-property-value?
3500 </property>
3501
3502 <component name="xs:NCName" autowire="xs:boolean"?
3503     requires="list of xs:QName"? policySets="list of xs:QName"?>*
3504 <implementation ... />?
3505 <service name="xs:NCName" requires="list of xs:QName"?
3506     policySets="list of xs:QName"?>*
3507 <interface ... />?
3508 <binding uri="xs:anyURI"? name="xs:NCName"?
3509     requires="list of xs:QName"?
3510     policySets="list of xs:QName"?/>*
3511 <callback>?
3512     <binding uri="xs:anyURI"? name="xs:NCName"?
3513         requires="list of xs:QName"?
3514         policySets="list of xs:QName"?/>+
3515 </callback>
3516 </service>
3517 <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
3518     source="xs:string"? file="xs:anyURI"?>*
3519     property-value
3520 </property>
3521 <reference name="xs:NCName" target="list of xs:anyURI"?
3522     autowire="xs:boolean"? wiredByImpl="xs:boolean"?
3523     requires="list of xs:QName"? policySets="list of xs:QName"?
3524     multiplicity="0..1 or 1..1 or 0..n or 1..n"?/>*
3525 <interface ... />?
3526 <binding uri="xs:anyURI"? name="xs:NCName"?
3527     requires="list of xs:QName"?
3528     policySets="list of xs:QName"?/>*
3529 <callback>?
3530     <binding uri="xs:anyURI"? name="xs:NCName"?
3531         requires="list of xs:QName"?
3532         policySets="list of xs:QName"?/>+
3533 </callback>
3534 </reference>
3535 </component>
3536
3537 <wire source="xs:anyURI" target="xs:anyURI" />*
3538
3539 </composite>

```

3540

B. XML Schemas

3541

B.1 sca.xsd

3542

3543

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

3544

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

3545

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

3546

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

3547

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

3548

3549

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

Formatted: English (U.S.)

3550

3551

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

3552

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

3553

3554

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

3555

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

3556

3557

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

3558

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

3559

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

3560

3561

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

3562

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

3563

3564

```
</schema>
```

3565

3566

B.2 sca-core.xsd

3567

3568

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

3569

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

3570

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

3571

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

3572

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

3573

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

3574

3575

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

3576

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

Formatted: English (U.S.)

3577

```
    <sequence>
```

3578

```
      <element ref="sca:implementation" minOccurs="0" maxOccurs="1"/>
```

```

3579     <choice minOccurs="0" maxOccurs="unbounded">
3580     <element name="service" type="sca:ComponentService" />
3581     <element name="reference" type="sca:ComponentReference"/>
3582     <element name="property" type="sca:Property"/>
3583     </choice>
3584     <any namespace="##other" processContents="lax" minOccurs="0"
3585         maxOccurs="unbounded" />
3586     </sequence>
3587     <attribute name="constrainingType" type="QName" use="optional"/>
3588     <anyAttribute namespace="##any" processContents="lax"/>
3589 </complexType>
3590
3591 <element name="composite" type="sca:Composite"/>
3592 <complexType name="Composite">
3593     <sequence>
3594         <element name="include" type="anyURI" minOccurs="0"
3595             maxOccurs="unbounded" />
3596         <choice minOccurs="0" maxOccurs="unbounded">
3597             <element name="service" type="sca:Service"/>
3598             <element name="property" type="sca:Property"/>
3599             <element name="component" type="sca:Component"/>
3600             <element name="reference" type="sca:Reference"/>
3601             <element name="wire" type="sca:Wire"/>
3602         </choice>
3603         <any namespace="##other" processContents="lax" minOccurs="0"
3604             maxOccurs="unbounded" />
3605     </sequence>
3606     <attribute name="name" type="NCName" use="required"/>
3607     <attribute name="targetNamespace" type="anyURI" use="required"/>
3608     <attribute name="local" type="boolean" use="optional"
3609 default="false"/>
3610     <attribute name="autowire" type="boolean" use="optional"
3611 default="false"/>
3612     <attribute name="constrainingType" type="QName" use="optional"/>
3613     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3614     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3615     <anyAttribute namespace="##any" processContents="lax"/>
3616 </complexType>
3617
3618 <complexType name="Service">
3619     <sequence>
3620         <element ref="sca:interface" minOccurs="0" maxOccurs="1" />
3621         <element name="operation" type="sca:Operation" minOccurs="0"

```

Formatted: English (U.S.)

Formatted: English (U.S.)

```

3622         maxOccurs="unbounded" />
3623     <choice minOccurs="0" maxOccurs="unbounded">
3624         <element ref="sca:binding" />
3625         <any namespace="##other" processContents="lax"
3626             minOccurs="0" maxOccurs="unbounded" />
3627     </choice>
3628     <element ref="sca:callback" minOccurs="0" maxOccurs="1" />
3629     <any namespace="##other" processContents="lax" minOccurs="0"
3630         maxOccurs="unbounded" />
3631 </sequence>
3632 <attribute name="name" type="NCName" use="required" />
3633 <attribute name="promote" type="anyURI" use="required" />
3634 <attribute name="requires" type="sca:listOfQNames" use="optional" />
3635 <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3636 <anyAttribute namespace="##any" processContents="lax" />
3637 </complexType>
3638
3639 <element name="interface" type="sca:Interface" abstract="true" />
3640 <complexType name="Interface" abstract="true"/>
3641
3642 <complexType name="Reference">
3643     <sequence>
3644         <element ref="sca:interface" minOccurs="0" maxOccurs="1" />
3645         <element name="operation" type="sca:Operation" minOccurs="0"
3646             maxOccurs="unbounded" />
3647         <choice minOccurs="0" maxOccurs="unbounded">
3648             <element ref="sca:binding" />
3649             <any namespace="##other" processContents="lax" />
3650         </choice>
3651         <element ref="sca:callback" minOccurs="0" maxOccurs="1" />
3652         <any namespace="##other" processContents="lax" minOccurs="0"
3653             maxOccurs="unbounded" />
3654     </sequence>
3655     <attribute name="name" type="NCName" use="required" />
3656     <attribute name="target" type="sca:listOfAnyURIs" use="optional"/>
3657     <attribute name="wiredByImpl" type="boolean" use="optional"
3658 default="false"/>
3659     <attribute name="multiplicity" type="sca:Multiplicity"
3660         use="optional" default="1..1" />
3661     <attribute name="promote" type="sca:listOfAnyURIs" use="required" />
3662     <attribute name="requires" type="sca:listOfQNames" use="optional" />
3663     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3664     <anyAttribute namespace="##any" processContents="lax" />

```

Formatted: French (France)

```

3665     </complexType>
3666
3667     <complexType name="SCAPropertyBase" mixed="true">
3668         <!-- mixed="true" to handle simple type -->
3669         <sequence>
3670             <any namespace="##any" processContents="lax" minOccurs="0"
3671                 maxOccurs="1" />
3672             <!-- NOT an extension point; This xsd:any exists to accept
3673                 the element-based or complex type property
3674                 i.e. no element-based extension point under "sca:property"
3675     -->
3676         </sequence>
3677     </complexType>
3678
3679     <!-- complex type for sca:property declaration -->
3680     <complexType name="Property" mixed="true">
3681         <complexContent>
3682             <extension base="sca:SCAPropertyBase">
3683                 <!-- extension defines the place to hold default value -->
3684                 <attribute name="name" type="NCName" use="required"/>
3685                 <attribute name="type" type="QName" use="optional"/>
3686                 <attribute name="element" type="QName" use="optional"/>
3687                 <attribute name="many" type="boolean" default="false"
3688                     use="optional"/>
3689                 <attribute name="mustSupply" type="boolean" default="false"
3690                     use="optional"/>
3691                 <anyAttribute namespace="##any" processContents="lax"/>
3692                 <!-- an extension point ; attribute-based only -->
3693             </extension>
3694         </complexContent>
3695     </complexType>
3696
3697     <complexType name="PropertyValue" mixed="true">
3698         <complexContent>
3699             <extension base="sca:SCAPropertyBase">
3700                 <attribute name="name" type="NCName" use="required"/>
3701                 <attribute name="type" type="QName" use="optional"/>
3702                 <attribute name="element" type="QName" use="optional"/>
3703                 <attribute name="many" type="boolean" default="false"
3704                     use="optional"/>
3705                 <attribute name="source" type="string" use="optional"/>
3706                 <attribute name="file" type="anyURI" use="optional"/>
3707                 <anyAttribute namespace="##any" processContents="lax"/>

```

```

3708         <!-- an extension point ; attribute-based only -->
3709         </extension>
3710     </complexContent>
3711 </complexType>
3712
3713 <element name="binding" type="sca:Binding" abstract="true"/>
3714 <complexType name="Binding" abstract="true">
3715     <sequence>
3716         <element name="operation" type="sca:Operation" minOccurs="0"
3717             maxOccurs="unbounded" />
3718     </sequence>
3719     <attribute name="uri" type="anyURI" use="optional"/>
3720     <attribute name="name" type="NCName" use="optional"/>
3721     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3722     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3723 </complexType>
3724
3725 <element name="bindingType" type="sca:BindingType"/>
3726 <complexType name="BindingType">
3727     <sequence minOccurs="0" maxOccurs="unbounded">
3728         <any namespace="##other" processContents="lax" />
3729     </sequence>
3730     <attribute name="type" type="QName" use="required"/>
3731     <attribute name="alwaysProvides" type="sca:listOfQNames"
3732 use="optional"/>
3733     <attribute name="mayProvide" type="sca:listOfQNames" use="optional"/>
3734     <anyAttribute namespace="##any" processContents="lax"/>
3735 </complexType>
3736
3737 <element name="callback" type="sca:Callback"/>
3738 <complexType name="Callback">
3739     <choice minOccurs="0" maxOccurs="unbounded">
3740         <element ref="sca:binding"/>
3741         <any namespace="##other" processContents="lax"/>
3742     </choice>
3743     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3744     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3745     <anyAttribute namespace="##any" processContents="lax"/>
3746 </complexType>
3747
3748 <complexType name="Component">
3749     <sequence>
3750         <element ref="sca:implementation" minOccurs="0" maxOccurs="1"/>

```

Formatted: English (U.S.)

```

3751         <choice minOccurs="0" maxOccurs="unbounded">
3752             <element name="service" type="sca:ComponentService"/>
3753             <element name="reference" type="sca:ComponentReference"/>
3754             <element name="property" type="sca:PropertyValue" />
3755         </choice>
3756         <any namespace="##other" processContents="lax" minOccurs="0"
3757             maxOccurs="unbounded" />
3758     </sequence>
3759     <attribute name="name" type="NCName" use="required" />
3760     <attribute name="autowire" type="boolean" use="optional" />
3761     <attribute name="constrainingType" type="QName" use="optional" />
3762     <attribute name="requires" type="sca:listOfQNames" use="optional" />
3763     <attribute name="policySets" type="sca:listOfQNames" use="optional" />
3764     <anyAttribute namespace="##any" processContents="lax" />
3765 </complexType>
3766
3767 <complexType name="ComponentService">
3768     <complexContent>
3769         <restriction base="sca:Service">
3770             <sequence>
3771                 <element ref="sca:interface" minOccurs="0"
3772 maxOccurs="1" />
3773                 <element name="operation" type="sca:Operation"
3774 minOccurs="0"
3775                 maxOccurs="unbounded" />
3776                 <choice minOccurs="0" maxOccurs="unbounded">
3777                     <element ref="sca:binding" />
3778                     <any namespace="##other" processContents="lax"
3779 minOccurs="0" maxOccurs="unbounded" />
3780                 </choice>
3781                 <element ref="sca:callback" minOccurs="0"
3782 maxOccurs="1" />
3783                 <any namespace="##other" processContents="lax"
3784 minOccurs="0"
3785                 maxOccurs="unbounded" />
3786             </sequence>
3787             <attribute name="name" type="NCName" use="required" />
3788             <attribute name="requires" type="sca:listOfQNames"
3789 use="optional" />
3790             <attribute name="policySets" type="sca:listOfQNames"
3791 use="optional" />
3792             <anyAttribute namespace="##any" processContents="lax" />
3793         </restriction>
3794     </complexContent>

```

Formatted: English (U.S.)

```

3795     </complexType>
3796
3797     <complexType name="ComponentReference">
3798         <complexContent>
3799             <restriction base="sca:Reference">
3800                 <sequence>
3801                     <element ref="sca:interface" minOccurs="0"
3802 maxOccurs="1" />
3803                     <element name="operation" type="sca:Operation"
3804 minOccurs="0"
3805                         maxOccurs="unbounded" />
3806                     <choice minOccurs="0" maxOccurs="unbounded">
3807                         <element ref="sca:binding" />
3808                         <any namespace="##other" processContents="lax"
3809 />
3810                     </choice>
3811                     <element ref="sca:callback" minOccurs="0"
3812 maxOccurs="1" />
3813                     <any namespace="##other" processContents="lax"
3814 minOccurs="0"
3815                         maxOccurs="unbounded" />
3816                     </sequence>
3817                     <attribute name="name" type="NCName" use="required" />
3818                     <attribute name="autowire" type="boolean" use="optional" />
3819                     <attribute name="wiredByImpl" type="boolean" use="optional"
3820                         default="false"/>
3821                     <attribute name="target" type="sca:listOfAnyURIs"
3822 use="optional"/>
3823                     <attribute name="multiplicity" type="sca:Multiplicity"
3824                         use="optional" default="1..1" />
3825                     <attribute name="requires" type="sca:listOfQNames"
3826 use="optional"/>
3827                     <attribute name="policySets" type="sca:listOfQNames"
3828                         use="optional"/>
3829                     <anyAttribute namespace="##any" processContents="lax" />
3830                 </restriction>
3831             </complexContent>
3832         </complexType>
3833
3834         <element name="implementation" type="sca:Implementation"
3835             abstract="true" />
3836         <complexType name="Implementation" abstract="true">
3837             <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3838             <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3839         </complexType>

```

```

3840
3841 <element name="implementationType" type="sca:ImplementationType"/>
3842 <complexType name="ImplementationType">
3843   <sequence minOccurs="0" maxOccurs="unbounded">
3844     <any namespace="##other" processContents="lax" />
3845   </sequence>
3846   <attribute name="type" type="QName" use="required"/>
3847   <attribute name="alwaysProvides" type="sca:listOfQNames"
3848 use="optional"/>
3849   <attribute name="mayProvide" type="sca:listOfQNames" use="optional"/>
3850   <anyAttribute namespace="##any" processContents="lax"/>
3851 </complexType>
3852
3853 <complexType name="Wire">
3854   <sequence>
3855     <any namespace="##other" processContents="lax" minOccurs="0"
3856       maxOccurs="unbounded" />
3857   </sequence>
3858   <attribute name="source" type="anyURI" use="required"/>
3859   <attribute name="target" type="anyURI" use="required"/>
3860   <anyAttribute namespace="##any" processContents="lax"/>
3861 </complexType>
3862
3863 <element name="include" type="sca:Include"/>
3864 <complexType name="Include">
3865   <attribute name="name" type="QName"/>
3866   <anyAttribute namespace="##any" processContents="lax"/>
3867 </complexType>
3868
3869 <complexType name="Operation">
3870   <attribute name="name" type="NCName" use="required"/>
3871   <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3872   <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3873   <anyAttribute namespace="##any" processContents="lax"/>
3874 </complexType>
3875
3876 <element name="constrainingType" type="sca:ConstrainingType"/>
3877 <complexType name="ConstrainingType">
3878   <sequence>
3879     <choice minOccurs="0" maxOccurs="unbounded">
3880       <element name="service" type="sca:ComponentService"/>
3881       <element name="reference" type="sca:ComponentReference"/>
3882       <element name="property" type="sca:Property" />

```

Formatted: French (France)

```

3883         </choice>
3884         <any namespace="##other" processContents="lax" minOccurs="0"
3885             maxOccurs="unbounded" />
3886     </sequence>
3887     <attribute name="name" type="NCName" use="required" />
3888     <attribute name="targetNamespace" type="anyURI" />
3889     <attribute name="requires" type="sca:listOfQNames" use="optional" />
3890     <anyAttribute namespace="##any" processContents="lax" />
3891 </complexType>
3892
3893
3894 <simpleType name="Multiplicity">
3895     <restriction base="string">
3896         <enumeration value="0..1" />
3897         <enumeration value="1..1" />
3898         <enumeration value="0..n" />
3899         <enumeration value="1..n" />
3900     </restriction>
3901 </simpleType>
3902
3903 <simpleType name="OverrideOptions">
3904     <restriction base="string">
3905         <enumeration value="no" />
3906         <enumeration value="may" />
3907         <enumeration value="must" />
3908     </restriction>
3909 </simpleType>
3910
3911 <!-- Global attribute definition for @requires to permit use of intents
3912     within WSDL documents -->
3913 <attribute name="requires" type="sca:listOfQNames" />
3914
3915 <!-- Global attribute definition for @endsConversation to mark operations
3916     as ending a conversation -->
3917 <attribute name="endsConversation" type="boolean" default="false" />
3918
3919 <simpleType name="listOfQNames">
3920     <list itemType="QName" />
3921 </simpleType>
3922
3923 <simpleType name="listOfAnyURIs">
3924     <list itemType="anyURI" />
3925 </simpleType>

```

3926
3927 </schema>

3928 **B.3 sca-binding-sca.xsd**

3929
3930 <?xml version="1.0" encoding="UTF-8"?>
3931 <!-- (c) Copyright SCA Collaboration 2006, 2007 -->
3932 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3933 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3934 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3935 elementFormDefault="qualified">
3936
3937 <include schemaLocation="sca-core.xsd"/>
3938
3939 <element name="binding.sca" type="sca:SCABinding"
3940 substitutionGroup="sca:binding"/>
3941 <complexType name="SCABinding">
3942 <complexContent>
3943 <extension base="sca:Binding">
3944 <sequence>
3945 <element name="operation" type="sca:Operation"
3946 minOccurs="0"
3947 maxOccurs="unbounded" />
3948 </sequence>
3949 <attribute name="uri" type="anyURI" use="optional"/>
3950 <attribute name="name" type="QName" use="optional"/>
3951 <attribute name="requires" type="sca:listOfQNames"
3952 use="optional"/>
3953 <attribute name="policySets" type="sca:listOfQNames"
3954 use="optional"/>
3955 <anyAttribute namespace="##any" processContents="lax"/>
3956 </extension>
3957 </complexContent>
3958 </complexType>
3959 </schema>
3960

Formatted: English (U.S.)

Formatted: English (U.S.)

3961 **B.4 sca-interface-java.xsd**

3962
3963 <?xml version="1.0" encoding="UTF-8"?>
3964 <!-- (c) Copyright SCA Collaboration 2006 -->
3965 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3966 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"

```

3967     xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3968     ▲elementFormDefault="qualified">
3969
3970     <include schemaLocation="sca-core.xsd"/>
3971
3972     ▲element name="interface.java" type="sca:JavaInterface"
3973           substitutionGroup="sca:interface"/>
3974     <complexType name="JavaInterface">
3975       <complexContent>
3976         <extension base="sca:Interface">
3977           <sequence>
3978             <any namespace="##other" processContents="lax"
3979 minOccurs="0"
3980               maxOccurs="unbounded"/>
3981           </sequence>
3982           <attribute name="interface" type="NCName" use="required"/>
3983           <attribute name="callbackInterface" type="NCName"
3984 use="optional"/>
3985           <anyAttribute namespace="##any" processContents="lax"/>
3986         </extension>
3987       </complexContent>
3988     </complexType>
3989 </schema>

```

Formatted: English (U.S.)

Formatted: French (France)

3990 B.5 sca-interface-wsdl.xsd

```

3991
3992 <?xml version="1.0" encoding="UTF-8"?>
3993 <!-- (c) Copyright SCA Collaboration 2006 -->
3994 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3995   targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3996   xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3997   ▲elementFormDefault="qualified">
3998
3999   <include schemaLocation="sca-core.xsd"/>
4000
4001   <element name="interface.wsdl" type="sca:WSDLPortType"
4002         substitutionGroup="sca:interface"/>
4003   <complexType name="WSDLPortType">
4004     ▲<complexContent>
4005       <extension base="sca:Interface">
4006         ▲<sequence>
4007           <any namespace="##other" processContents="lax"
4008 minOccurs="0"
4009             maxOccurs="unbounded"/>
4010         </sequence>
4011         <attribute name="interface" type="anyURI" use="required"/>

```

Formatted: English (U.S.)

Formatted: French (France)

Formatted: English (U.S.)

```

4011         <attribute name="callbackInterface" type="anyURI"
4012 use="optional"/>
4013         <anyAttribute namespace="##any" processContents="lax"/>
4014     </extension>
4015 </complexContent>
4016 </complexType>
4017 </schema>
4018

```

4019 B.6 sca-implementation-java.xsd

```

4020
4021 <?xml version="1.0" encoding="UTF-8"?>
4022 <!-- (c) Copyright SCA Collaboration 2006 -->
4023 <schema xmlns="http://www.w3.org/2001/XMLSchema"
4024 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4025 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4026 elementFormDefault="qualified">
4027
4028     <include schemaLocation="sca-core.xsd"/>
4029
4030     <element name="implementation.java" type="sca:JavaImplementation"
4031 substitutionGroup="sca:implementation"/>
4032     <complexType name="JavaImplementation">
4033         <complexContent>
4034             <extension base="sca:Implementation">
4035                 <sequence>
4036                     <any namespace="##other" processContents="lax"
4037 minOccurs="0" maxOccurs="unbounded"/>
4038                 </sequence>
4039                 <attribute name="class" type="NCName" use="required"/>
4040                 <attribute name="requires" type="sca:listOfQNames"
4041 use="optional"/>
4042                 <attribute name="policySets" type="sca:listOfQNames"
4043 use="optional"/>
4044                 <anyAttribute namespace="##any" processContents="lax"/>
4045             </extension>
4046         </complexContent>
4047     </complexType>
4048 </schema>

```

Formatted: English (U.S.)

Formatted: French (France)

4049 B.7 sca-implementation-composite.xsd

```

4050
4051 <?xml version="1.0" encoding="UTF-8"?>

```

```

4052 <!-- (c) Copyright SCA Collaboration 2006 -->
4053 <schema xmlns="http://www.w3.org/2001/XMLSchema"
4054     targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4055     xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4056     elementFormDefault="qualified">
4057
4058     <include schemaLocation="sca-core.xsd"/>
4059     <element name="implementation.composite" type="sca:SCAImplementation"
4060         substitutionGroup="sca:implementation"/>
4061     <complexType name="SCAImplementation">
4062         <complexContent>
4063             <extension base="sca:Implementation">
4064                 <sequence>
4065                     <any namespace="##other" processContents="lax"
4066 minOccurs="0"
4067                         maxOccurs="unbounded"/>
4068                 </sequence>
4069                 <attribute name="name" type="QName" use="required"/>
4070                 <attribute name="requires" type="sca:listOfQNames"
4071 use="optional"/>
4072                 <attribute name="policySets" type="sca:listOfQNames"
4073 use="optional"/>
4074                 <anyAttribute namespace="##any" processContents="lax"/>
4075             </extension>
4076         </complexContent>
4077     </complexType>
4078 </schema>
4079

```

Formatted: English (U.S.)

Formatted: French (France)

4080 B.8 sca-definitions.xsd

```

4081
4082 <?xml version="1.0" encoding="UTF-8"?>
4083 <!-- (c) Copyright SCA Collaboration 2006 -->
4084 <schema xmlns="http://www.w3.org/2001/XMLSchema"
4085     targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4086     xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4087     elementFormDefault="qualified">
4088
4089     <include schemaLocation="sca-core.xsd"/>
4090
4091     <element name="definitions">
4092         <complexType>
4093             <choice minOccurs="0" maxOccurs="unbounded">

```

```
4094         <element ref="sca:intent" />
4095         <element ref="sca:policySet" />
4096         <element ref="sca:binding" />
4097         <element ref="sca:bindingType" />
4098         <element ref="sca:implementationType" />
4099         <any namespace="##other" processContents="lax" minOccurs="0"
4100             maxOccurs="unbounded" />
4101     </choice>
4102 </complexType>
4103 </element>
4104
4105 </schema>
4106
```

4107 **B.9 sca-binding-webservice.xsd**

4108 Is described in [the SCA Web Services Binding specification \[9\]](#)

4109 **B.10 sca-binding-jms.xsd**

4110 Is described in [the SCA JMS Binding specification \[11\]](#)

4111 **B.11 sca-policy.xsd**

4112 Is described in [the SCA Policy Framework specification \[10\]](#)

4113 C. SCA Concepts

4114 C.1 Binding

4115 **Bindings** are used by services and references. References use bindings to describe the access
4116 mechanism used to call the service to which they are wired. Services use bindings to describe the
4117 access mechanism(s) that clients should use to call the service.

4118 SCA supports multiple different types of bindings. Examples include **SCA service, Web service,**
4119 **stateless session EJB, data base stored procedure, EIS service.** SCA provides an extensibility
4120 mechanism by which an SCA runtime can add support for additional binding types.

4121

4122 C.2 Component

4123 **SCA components** are configured instances of **SCA implementations**, which provide and consume
4124 services. SCA allows many different implementation technologies such as Java, BPEL, C++. SCA defines
4125 an **extensibility mechanism** that allows you to introduce new implementation types. The current
4126 specification does not mandate the implementation technologies to be supported by an SCA run-time,
4127 vendors may choose to support the ones that are important for them. A single SCA implementation may
4128 be used by multiple Components, each with a different configuration.

4129 The Component has a reference to an implementation of which it is an instance, a set of property values,
4130 and a set of service reference values. Property values define the values of the properties of the
4131 component as defined by the component's implementation. Reference values define the services that
4132 resolve the references of the component as defined by its implementation. These values can either be a
4133 particular service of a particular component, or a reference of the containing composite.

4134 C.3 Service

4135 **SCA services** are used to declare the externally accessible services of an **implementation**. For a
4136 composite, a service is typically provided by a service of a component within the composite, or by a
4137 reference defined by the composite. The latter case allows the republication of a service with a new
4138 address and/or new bindings. The service can be thought of as a point at which messages from external
4139 clients enter a composite or implementation.

4140 A service represents an addressable set of operations of an implementation that are designed to be
4141 exposed for use by other implementations or exposed publicly for use elsewhere (eg public Web services
4142 for use by other organizations). The operations provided by a service are specified by an Interface, as
4143 are the operations required by the service client (if there is one). An implementation may contain
4144 multiple services, when it is possible to address the services of the implementation separately.

4145 A service may be provided **as SCA remote services, as Web services, as stateless session EJB's, as**
4146 **EIS services, and so on.** Services use **bindings** to describe the way in which they are published. SCA
4147 provides an **extensibility mechanism** that makes it possible to introduce new binding types for new
4148 types of services.

4149 C.3.1 Remotable Service

4150 A Remotable Service is a service that is designed to be published remotely in a loosely-coupled
4151 SOA architecture. For example, SCA services of SCA implementations can define
4152 implementations of industry-standard web services. Remotable services use pass-by-value
4153 semantics for parameters and returned results.

4154 A service is remotable if it is defined by a WSDL port type or if it defined by a Java interface
4155 marked with the @Remotable annotation.

4156 C.3.2 Local Service

4157 Local services are services that are designed to be only used “locally” by other implementations
4158 that are deployed concurrently in a tightly-coupled architecture within the same operating system
4159 process.

4160 Local services may rely on by-reference calling conventions, or may assume a very fine-grained
4161 interaction style that is incompatible with remote distribution. They may also use technology-
4162 specific data-types.

4163 Currently a service is local only if it defined by a Java interface not marked with the @Remotable
4164 annotation.

4165

4166 C.4 Reference

4167 **SCA references** represent a dependency that an implementation has on a service that is supplied by
4168 some other implementation, where the service to be used is specified through configuration. In other
4169 words, a reference is a service that an implementation may call during the execution of its business
4170 function. References are typed by an interface.

4171 For composites, composite references can be accessed by components within the composite like any
4172 service provided by a component within the composite. Composite references can be used as the targets
4173 of wires from component references when configuring Components.

4174 A composite reference can be used to access a service such as: an SCA service provided by another
4175 SCA composite, a Web service, a stateless session EJB, a data base stored procedure or an EIS service,
4176 and so on. References use **bindings** to describe the access method used to their services. SCA provides
4177 an **extensibility mechanism** that allows the introduction of new binding types to references.

4178

4179 C.5 Implementation

4180 An implementation is concept that is used to describe a piece of software technology such as a Java
4181 class, BPEL process, XSLT transform, or C++ class that is used to implement one or more services in a
4182 service-oriented application. An SCA composite is also an implementation.

4183 Implementations define points of variability including properties that can be set and settable references to
4184 other services. The points of variability are configured by a component that uses the implementation. The
4185 specification refers to the configurable aspects of an implementation as its **componentType**.

4186 C.6 Interface

4187 **Interfaces** define one or more business functions. These business functions are provided by Services
4188 and are used by components through References. Services are defined by the Interface they implement.
4189 SCA currently supports two interface type systems:

- 4190 • Java interfaces
- 4191 • WSDL portTypes

4192

4193 SCA also provides an extensibility mechanism by which an SCA runtime can add support for additional
4194 interface type systems.

4195 Interfaces may be **bi-directional**. A bi-directional service has service operations which must be provided
4196 by each end of a service communication – this could be the case where a particular service requires a
4197 “callback” interface on the client, which is calls during the process of handing service requests from the
4198 client.

4199

4200 C.7 Composite

4201 An SCA composite is the basic unit of composition within an SCA Domain. An **SCA Composite** is an
4202 assembly of Components, Services, References, and the Wires that interconnect them. Composites can
4203 be used to contribute elements to an **SCA Domain**.

4204 A **composite** has the following characteristics:

- 4205 • It may be used as a component implementation. When used in this way, it defines a boundary for
4206 Component visibility. Components may not be directly referenced from outside of the composite
4207 in which they are declared.
- 4208 • It can be used to define a unit of deployment. Composites are used to contribute business logic
4209 artifacts to an SCA domain.

4210

4211 C.8 Composite inclusion

4212 One composite can be used to provide part of the definition of another composite, through the process of
4213 inclusion. This is intended to make team development of large composites easier. Included composites
4214 are merged together into the using composite at deployment time to form a single logical composite.

4215 Composites are included into other composites through `<include.../>` elements in the using composite.
4216 The SCA Domain uses composites in a similar way, through the deployment of composite files to a
4217 specific location.

4218

4219 C.9 Property

4220 **Properties** allow for the configuration of an implementation with externally set data values. The data
4221 value is provided through a Component, possibly sourced from the property of a containing composite.

4222 Each Property is defined by the implementation. Properties may be defined directly through the
4223 implementation language or through annotations of implementations, where the implementation language
4224 permits, or through a componentType file. A Property can be either a simple data type or a complex data
4225 type. For complex data types, XML schema is the preferred technology for defining the data types.

4226

4227 C.10 Domain

4228 An SCA Domain represents a set of Services providing an area of Business functionality that is controlled
4229 by a single organization. As an example, for the accounts department in a business, the SCA Domain
4230 might cover all finance-related functions, and it might contain a series of composites dealing with specific
4231 areas of accounting, with one for Customer accounts, another dealing with Accounts Payable.

4232 A domain specifies the instantiation, configuration and connection of a set of components, provided via
4233 one or more composite files. The domain, like a composite, also has Services and References. Domains
4234 also contain Wires which connect together the Components, Services and References.

4235

4236 C.11 Wire

4237 **SCA wires** connect **service references** to **services**.

4238 Within a composite, valid wire sources are component references and composite services. Valid wire
4239 targets are component services and composite references.

4240 When using included composites, the sources and targets of the wires don't have to be declared in the
4241 same composite as the composite that contains the wire. The sources and targets can be defined by
4242 other included composites. Targets can also be external to the SCA domain.

4243

4244 **D. Acknowledgements**

4245 The following individuals have participated in the creation of this specification and are gratefully
4246 acknowledged:

4247 **Participants:**

4248 [Participant Name, Affiliation | Individual Member]

4249 [Participant Name, Affiliation | Individual Member]

4250

E. Non-Normative Text

4252

F. Revision History

4253

[optional; should not be included in OASIS Standards]

4254

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

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