



Service Component Architecture Assembly Model Specification Version 1.1

Committee Draft 01

18th March, 2008

Specification URIs:

This Version:

- <http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-CD-01.html>
- <http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-CD-01.doc>
- <http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-CD-01.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.....	7
1.1	Terminology.....	7
1.2	Normative References.....	7
2	Overview.....	9
2.1	Diagram used to Represent SCA Artifacts.....	10
3	Quick Tour by Sample.....	12
4	Implementation and ComponentType.....	13
4.1	Component Type.....	13
4.1.1	Service.....	14
4.1.2	Reference.....	15
4.1.3	Property.....	17
4.1.4	Implementation.....	18
4.2	Example ComponentType.....	19
4.3	Example Implementation.....	19
5	Component.....	22
5.1	Implementation.....	23
5.2	Service.....	24
5.3	Reference.....	25
5.4	Property.....	27
5.5	Example Component.....	29
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

9	Binding.....	71
9.1	Messages containing Data not defined in the Service Interface	73
9.2	Form of the URI of a Deployed Binding.....	73
9.2.1	Constructing Hierarchical URIs	73
9.2.2	Non-hierarchical URIs	74
9.2.3	Determining the URI scheme of a deployed binding.....	74
9.3	SCA Binding	75
9.3.1	Example SCA Binding	75
9.4	Web Service Binding	76
9.5	JMS Binding.....	76
10	SCA Definitions	78
11	Extension Model.....	80
11.1	Defining an Interface Type.....	80
11.2	Defining an Implementation Type.....	82
11.3	Defining a Binding Type.....	83
12	Packaging and Deployment	87
12.1	Domains.....	87
12.2	Contributions.....	87
12.2.1	SCA Artifact Resolution.....	88
12.2.2	SCA Contribution Metadata Document	89
12.2.3	Contribution Packaging using ZIP.....	90
12.3	Installed Contribution	91
12.3.1	Installed Artifact URIs.....	91
12.4	Operations for Contributions.....	91
12.4.1	install Contribution & update Contribution.....	92
12.4.2	add Deployment Composite & update Deployment Composite.....	92
12.4.3	remove Contribution	92
12.5	Use of Existing (non-SCA) Mechanisms for Resolving Artifacts	92
12.6	Domain-Level Composite	93
12.6.1	add To Domain-Level Composite.....	93
12.6.2	remove From Domain-Level Composite	93
12.6.3	get Domain-Level Composite	93
12.6.4	get QName Definition	93
13	Conformance	94
A.	Pseudo Schema	95
A.1	ComponentType	95
A.2	Composite	96
B.	XML Schemas	98
B.1	sca.xsd	98
B.2	sca-core.xsd	98
B.3	sca-binding-sca.xsd.....	107
B.4	sca-interface-java.xsd	107
B.5	sca-interface-wsdl.xsd.....	108
B.6	sca-implementation-java.xsd.....	109
B.7	sca-implementation-composite.xsd.....	109

B.8 sca-definitions.xsd	110
B.9 sca-binding-webservice.xsd	111
B.10 sca-binding-jms.xsd.....	111
B.11 sca-policy.xsd.....	111
C. SCA Concepts	113
C.1 Binding	113
C.2 Component.....	113
C.3 Service	113
C.3.1 Remotable Service	113
C.3.2 Local Service	114
C.4 Reference.....	114
C.5 Implementation.....	114
C.6 Interface	114
C.7 Composite	115
C.8 Composite inclusion	115
C.9 Property.....	115
C.10 Domain	115
C.11 Wire.....	115
D. Acknowledgements	117
E. Non-Normative Text	118
F. Revision History.....	119

1 Introduction

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

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

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

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

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

1.1 Terminology

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

1.2 Normative References

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

[1] SCA Java Component Implementation Specification

SCA Java Common Annotations and APIs Specification

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

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

[2] SDO Specification

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

[3] SCA Example Code document

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

[4] JAX-WS Specification

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

[5] WS-I Basic Profile

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

[6] WS-I Basic Security Profile

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

[7] Business Process Execution Language (BPEL)

http://www.oasis-open.org/committees/documents.php?wg_abbrev=wsbpel

- 39
- 40 [8] WSDL Specification
- 41 WSDL 1.1: <http://www.w3.org/TR/wsdl>
- 42 WSDL 2.0: <http://www.w3.org/TR/wsdl20/>
- 43
- 44 [9] SCA Web Services Binding Specification
- 45 http://www.osoa.org/download/attachments/35/SCA_WebServiceBindings_V100.pdf
- 46
- 47 [10] SCA Policy Framework Specification
- 48 http://www.osoa.org/download/attachments/35/SCA_Policy_Framework_V100.pdf
- 49
- 50 [11] SCA JMS Binding Specification
- 51 http://www.osoa.org/download/attachments/35/SCA_JMSBinding_V100.pdf
- 52
- 53 [12] ZIP Format Definition
- 54 <http://www.pkware.com/documents/casestudies/APPNOTE.TXT>
- 55
- 56 **[Reference]** [Full reference citation]

2 Overview

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

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

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

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

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

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

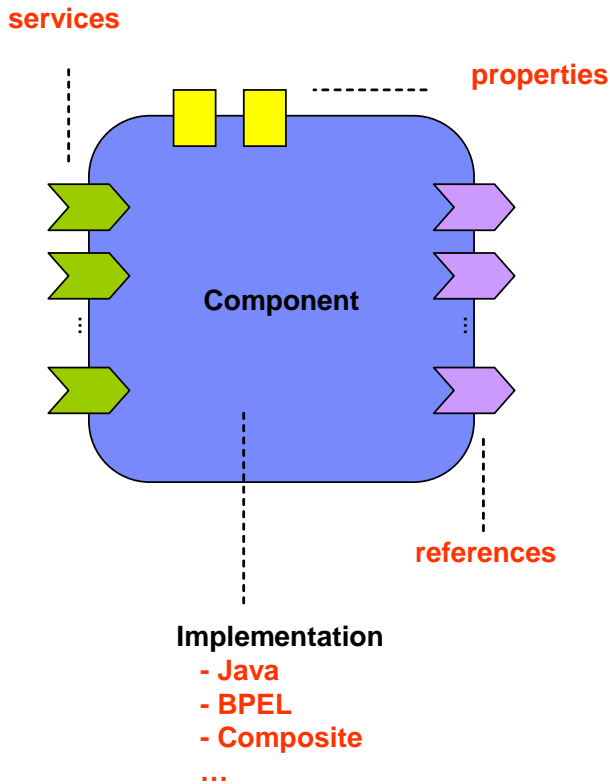
95 Composites are deployed within an **SCA Domain**. An SCA Domain typically represents a set of services
96 providing an area of business functionality that is controlled by a single organization. As an example, for
97 the accounts department in a business, the SCA Domain might cover all financial related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

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

107 **2.1 Diagram used to Represent SCA Artifacts**

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

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



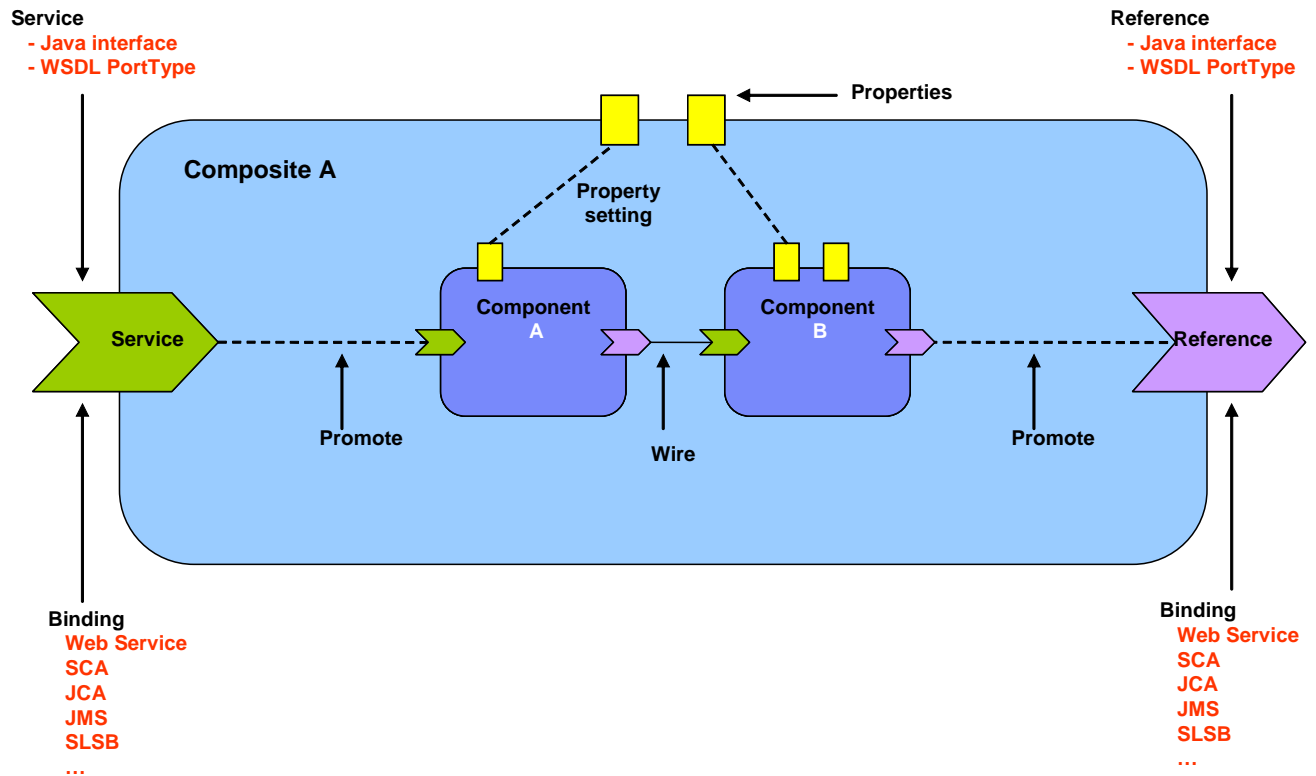
112

113 *Figure 1: SCA Component Diagram*

114

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

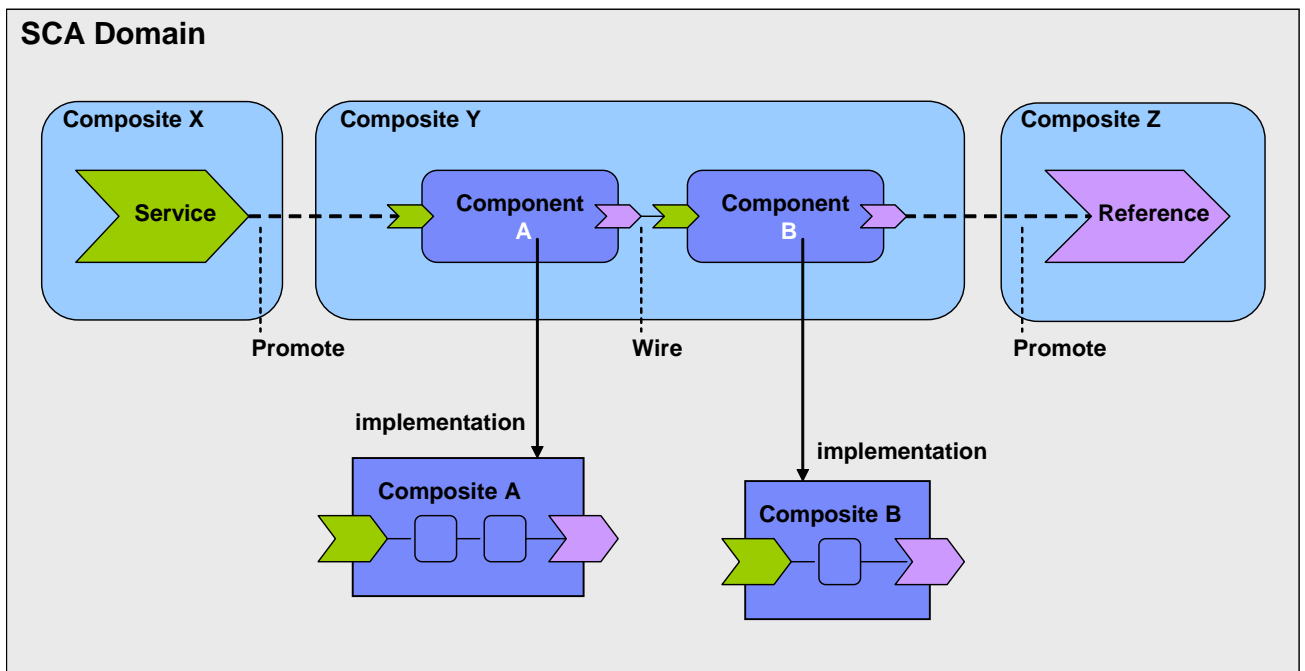
117



118
119 *Figure 2: SCA Composite Diagram*

120


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



123
124 *Figure 3: SCA Domain Diagram*

125 **3 Quick Tour by Sample**

126

127 ... 

128

129

4 Implementation and ComponentType

130

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

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

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

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

144 So, for example:

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

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

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

159

4.1 Component Type

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

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

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

177 A **component type file** has the same name as the implementation file but has the extension
178 ".**componentType**". The component type is defined by a **componentType element** in the file.
179 The **location** of the component type file depends on the type of the component implementation: it
180 is described in the respective client and implementation model specification for the implementation
181 type.

182 The following snippet shows the componentType schema.


```
183  
184 <?xml version="1.0" encoding="ASCII"?>  
185 <!-- Component type schema snippet -->  
186 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
187     constrainingType="QName"? >  
188  
189     <service ... />*  
190     <reference ... />*  
191     <property ... />*  
192     <implementation ... />?  
193  
194 </componentType>  
195
```

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

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

201

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

- 203 • **service : Service (0..n)** – see component type service section. 
- 204 • **reference : Reference (0..n)** – see component type reference section.
- 205 • **property : Property (0..n)** – see component type property section.
- 206 • **implementation : Implementation (0..1)** – see component type implementation
207 section.

208

209 4.1.1 Service

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

```
214  
215 <?xml version="1.0" encoding="ASCII"?>  
216 <!-- Component type service schema snippet -->  
217 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ...  
218 >  
219  
220     <service name="xs:NCName"
```

```

221         requires="list of xs:QName"? policySets="list of xs:QName"?>*
222     <interface ... />
223     <binding ... />*
224     <callback?
225         <binding ... />+
226     </callback>
227 </service>
228
229 <reference ... />*
230 <property ... />*
231 <implementation ... />?
232
233 </componentType>
234

```

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

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

241

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

- 243 • **interface** : *Interface (1..1)* - A service has **one interface**, which describes the operations provided by the service. The interface is described by an **interface element** which is a child element of the service element. For details on the interface element see [the Interface section](#).
- 244 • **binding** : *Binding (0..n)* - A service element has **zero or more binding elements** as children. If the binding element is not present it defaults to <binding.sca>. Details of the binding element are described in [the Bindings section](#). The binding, combined with any PolicySets in effect for the binding, must satisfy the set of policy intents for the service, as described in [the Policy Framework specification \[10\]](#).
- 245 • **callback (0..1) / binding** : *Binding (1..n)* - A service element has an optional **callback** element used if the interface has a callback defined, which has one or more **binding** elements as children. The **callback** and its binding child elements are specified if there is a need to have binding details used to handle callbacks. If the callback element is not present, the behaviour is runtime implementation dependent.

257

258 Reference

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

264

```

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

```

```

267 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ...
268 >
269
270 <service ... /*>
271
272 <reference name="xs:NCName"
273     target="list of xs:anyURI"? autowire="xs:boolean"?
274     multiplicity="0..1 or 1..1 or 0..n or 1..n"?
275     wiredByImpl="xs:boolean"?
276     requires="list of xs:QName"? policySets="list of xs:QName"?>*
277 <interface ... /*>
278 <binding ... /*>*
279 <callback?
280     <binding ... /*>+
281 </callback>
282 </reference>
283
284 <property ... /*>*
285 <implementation ... /*>?
286
287 </componentType>
288

```

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

- 290 • **name : NCName (1..1)** - the name of the reference.
- 291 • **multiplicity : 0..1|1..1|0..n|1..n (0..1)** - defines the number of wires that can connect
292 the reference to target services. The multiplicity can have the following values
 - 293 ○ 1..1 – one wire can have the reference as a source
 - 294 ○ 0..1 – zero or one wire can have the reference as a source
 - 295 ○ 1..n – one or more wires can have the reference as a source
 - 296 ○ 0..n - zero or more wires can have the reference as a source
- 297 • **target : anyURI (0..n)** - a list of one or more of target service URI's, depending on
298 multiplicity setting. Each value wires the reference to a component service that resolves
299 the reference. For more details on wiring see [the section on Wires](#).
- 300 • **autowire : boolean (0..1)** - whether the reference should be autowired, as described in
301 [the Autowire section](#). Default is false.
- 302 • **wiredByImpl : boolean (0..1)** - a boolean value, "false" by default, which indicates that
303 the implementation wires this reference dynamically. If set to "true" it indicates that the
304 target of the reference is set at runtime by the implementation code (eg by the code
305 obtaining an endpoint reference by some means and setting this as the target of the
306 reference through the use of programming interfaces defined by the relevant Client and
307 Implementation specification). If "true" is set, then the reference should not be wired
308 statically within a composite, but left unwired.
- 309 • **requires : QName (0..n)** - a list of policy intents. See the [Policy Framework specification](#)
310 [\[10\]](#) for a description of this attribute.
- 311 • **policySets : QName (0..n)** - a list of policy sets. See the [Policy Framework specification](#)
312 [\[10\]](#) for a description of this attribute.

313

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

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

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

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

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

336

337 4.1.3 Property

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

342

```
343 <?xml version="1.0" encoding="ASCII"?>
344 <!-- Component type property schema snippet -->
345 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ...
346 >
347
348     <service ... />*
349     <reference ... >*
350
351     <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
352         many="xs:boolean"? mustSupply="xs:boolean"?
353         policySets="list of xs:QName"?>* 
354         default-property-value?
355     </property>
356
357     <implementation ... />?
358
359 </componentType>
```

360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381

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

- **name : NCName (1..1)** - the name of the property.
- one of **(1..1)**:
 - **type : QName** - the type of the property defined as the qualified name of an XML schema type.
 - **element : QName** - the type of the property defined as the qualified name of an XML schema global element – the type is the type of the global element.
- **many : boolean (0..1)** - (optional) whether the property is single-valued (false) or multi-valued (true). In the case of a multi-valued property, it is presented to the implementation as a collection of property values.
- **mustSupply : boolean (0..1)** - whether the property value must be supplied by the component that uses the implementation – when mustSupply="true" the component must supply a value since the implementation has no default value for the property. A default-property-value should only be supplied when mustSupply="false" (the default setting for the mustSupply attribute), since the implication of a default value is that it is used only when a value is not supplied by the using component.
- **source : string (0..1)** - an XPath expression pointing to a property of the using composite from which the value of this property is obtained.
- **file : anyURI (0..1)** - a dereferencable URI to a file containing a value for the property.



4.1.4 Implementation

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

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

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

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

</componentType>
```

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

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

- 405
- **policySets** : *QName (0..n)* - a list of policy sets. See the [Policy Framework specification \[10\]](#) for a description of this attribute.
- 406
- 407



409 4.2 Example ComponentType

410

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

414

```
415 <?xml version="1.0" encoding="ASCII"?>
416 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
417
418     <service name="MyValueService">
419         <interface.java interface="services.myvalue.MyValueService"/>
420     </service>
421
422     <reference name="customerService">
423         <interface.java interface="services.customer.CustomerService"/>
424     </reference>
425     <reference name="stockQuoteService">
426         <interface.java
427 interface="services.stockquote.StockQuoteService"/>
428     </reference>
429
430     <property name="currency" type="xsd:string">USD</property>
431
432 </componentType>
433
```

434 4.3 Example Implementation

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

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

439

```
440 package services.account;
441
442 @Remotable
443 public interface AccountService{
444
445     public AccountReport getAccountReport(String customerID);
446 }
447
```

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

```
451  
452 package services.account;  
453  
454 import java.util.List;  
455  
456 import commonj.sdo.DataFactory;  
457  
458 import org.osoa.sca.annotations.Property;  
459 import org.osoa.sca.annotations.Reference;  
460  
461 import services.accountdata.AccountDataService;  
462 import services.accountdata.CheckingAccount;  
463 import services.accountdata.SavingsAccount;  
464 import services.accountdata.StockAccount;  
465 import services.stockquote.StockQuoteService;  
466  
467 public class AccountServiceImpl implements AccountService {  
468  
469     @Property  
470     private String currency = "USD";  
471  
472     @Reference  
473     private AccountDataService accountDataService;  
474     @Reference  
475     private StockQuoteService stockQuoteService;  
476  
477     public AccountReport getAccountReport(String customerID) {  
478  
479         DataFactory dataFactory = DataFactory.INSTANCE;  
480         AccountReport accountReport = (AccountReport)dataFactory.create(AccountReport.class);  
481         List accountSummaries = accountReport.getAccountSummaries();  
482  
483         CheckingAccount checkingAccount = accountDataService.getCheckingAccount(customerID);  
484         AccountSummary checkingAccountSummary =  
485 (AccountSummary)dataFactory.create(AccountSummary.class);  
486         checkingAccountSummary.setAccountNumber(checkingAccount.getAccountNumber());  
487         checkingAccountSummary.setAccountType("checking");  
488         checkingAccountSummary.setBalance(fromUSDollarToCurrency(checkingAccount.getBalance()));  
489         accountSummaries.add(checkingAccountSummary);  
490  
491         SavingsAccount savingsAccount = accountDataService.getSavingsAccount(customerID);  
492         AccountSummary savingsAccountSummary =  
493 (AccountSummary)dataFactory.create(AccountSummary.class);  
494         savingsAccountSummary.setAccountNumber(savingsAccount.getAccountNumber());  
495         savingsAccountSummary.setAccountType("savings");  
496         savingsAccountSummary.setBalance(fromUSDollarToCurrency(savingsAccount.getBalance()));  
497         accountSummaries.add(savingsAccountSummary);  
498
```

```

499         StockAccount stockAccount = accountDataService.getStockAccount(customerID);
500         AccountSummary stockAccountSummary =
501 (AccountSummary)dataFactory.create(AccountSummary.class);
502         stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());
503         stockAccountSummary.setAccountType("stock");
504         float balance=
505 (stockQuoteService.getQuote(stockAccount.getSymbol()))*stockAccount.getQuantity();
506         stockAccountSummary.setBalance(fromUSDollarToCurrency(balance));
507         accountSummaries.add(stockAccountSummary);
508
509         return accountReport;
510     }
511
512     private float fromUSDollarToCurrency(float value){
513
514         if (currency.equals("USD")) return value; else
515         if (currency.equals("EURO")) return value * 0.8f; else
516         return 0.0f;
517     }
518 }

```

519

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

522

```

523 <?xml version="1.0" encoding="ASCII"?>
524 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
525               xmlns:xsd="http://www.w3.org/2001/XMLSchema">
526
527     <service name="AccountService">
528         <interface.java interface="services.account.AccountService"/>
529     </service>
530     <reference name="accountDataService">
531         <interface.java
532 interface="services.accountdata.AccountDataService"/>
533     </reference>
534     <reference name="stockQuoteService">
535         <interface.java
536 interface="services.stockquote.StockQuoteService"/>
537     </reference>
538
539     <property name="currency" type="xsd:string">USD</property>
540
541 </componentType>
542

```

543

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

547 5 Component

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

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

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

557

```
558 <?xml version="1.0" encoding="UTF-8"?>
559 <!-- Component schema snippet -->
560 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
561   ...
562   <component name="xs:NCName" autowire="xs:boolean"?
563             requires="list of xs:QName"? policySets="list of xs:QName"?
564             constrainingType="xs:QName"?>*
565     <implementation ... />?
566     <service ... />*
567     <reference ... />*
568     <property ... />*
569   </component>
570   ...
571 </composite>
572
```

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

- 574 • **name : NCName (1..1)** – the name of the component. The name must be unique across
575 all the components in the composite.
- 576 • **autowire : boolean (0..1)** – whether contained component references should be
577 autowired, as described in [the Autowire section](#). Default is false.
- 578 • **requires : QName (0..n)** – a list of policy intents. See the [Policy Framework specification](#)
579 [\[10\]](#) for a description of this attribute.
- 580 • **policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification](#)
581 [\[10\]](#) for a description of this attribute.
- 582 • **constrainingType : QName (0..1)** – the name of a constrainingType. When specified,
583 the set of services, references and properties of the component, plus related intents, is
584 constrained to the set defined by the constrainingType. See [the ConstrainingType Section](#)
585 for more details.

586

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

- 588 • **implementation : ComponentImplementation (0..1)** – see component
589 implementation section. 

- 590 • **service** : *ComponentService (0..n)* – see component service section.
- 591 • **reference** : *ComponentReference (0..n)* – see component reference section.
- 592 • **property** : *ComponentProperty (0..n)* – see component property section.

593

594 5.1 Implementation

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

600

```
601 <?xml version="1.0" encoding="UTF-8"?>
602 <!-- Component Implementation schema snippet -->
603 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
604   ...
605   <component ... >*
606     <implementation ... />? 
607     <service ... />*
608     <reference ... />*
609     <property ... />*
610   </component>
611   ...
612 </composite>
```

613

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

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

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

623

```
624 <implementation.java class="services.myvalue.MyValueServiceImpl"/>
625
626 <implementation.bpel process="ans:MoneyTransferProcess"/>
627
628 <implementation.composite name="bns:MyValueComposite"/>
```

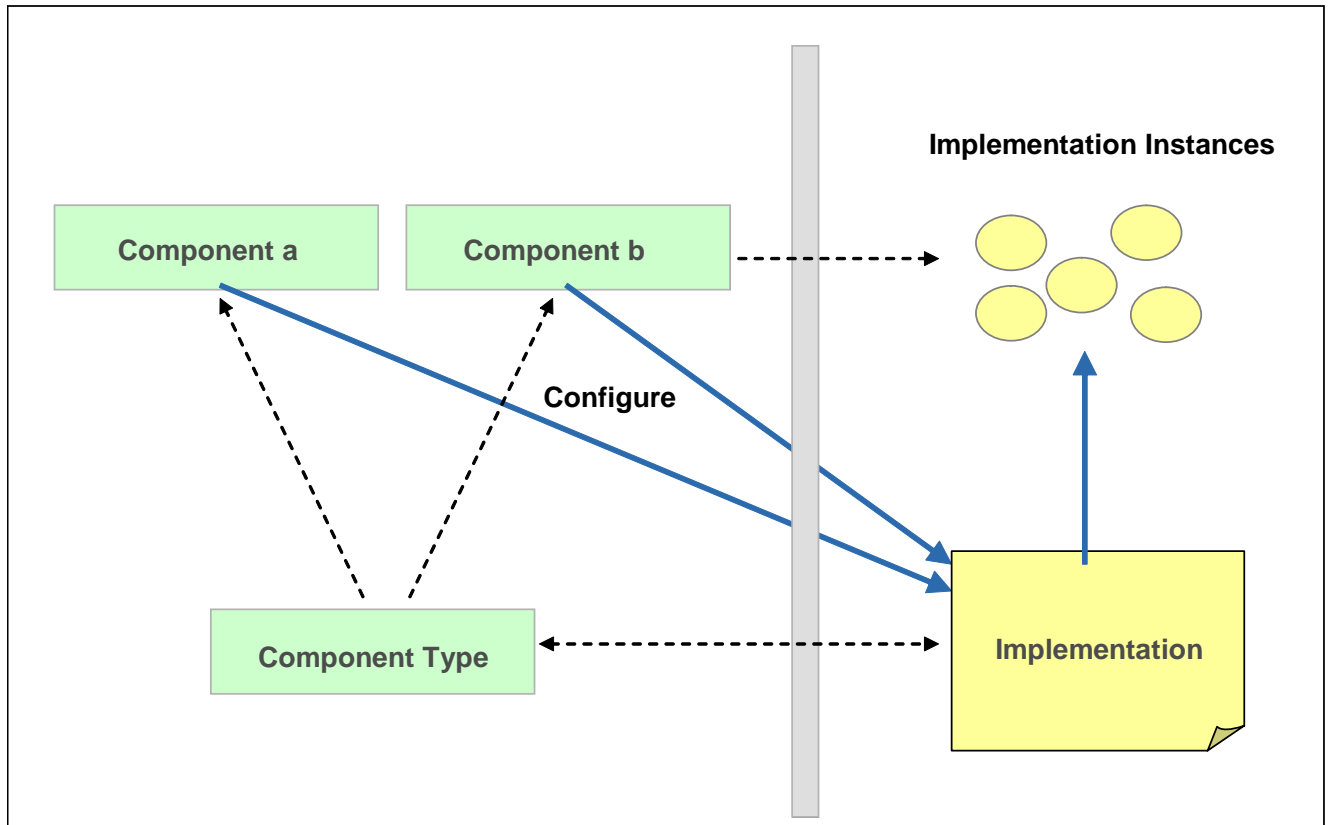
629

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

631

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

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



637
 638 *Figure 4: Relationship of Component and Implementation*

639

640 5.2 Service

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

645

```
646 <?xml version="1.0" encoding="UTF-8"?>
647 <!-- Component Service schema snippet -->
648 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
649   ...
650   <component ... >*
651     <implementation ... />?
652     <service name="xs:NCName" requires="list of xs:QName"?
653       policySets="list of xs:QName"?>*
654       <interface ... />?
655       <binding ... />*
656       <callback?>
```



```

657         <binding ... />+
658     </callback>
659 </service>
660 <reference ... />*
661 <property ... />*
662 </component>
663 ...
664 </composite>
665

```

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

- 667 • **name : NCName (1..1)** - the name of the service. Has to match a name of a service
668 defined by the implementation.
- 669 • **requires : QName (0..n)** – a list of policy intents. See the [Policy Framework specification](#)
670 [\[10\]](#) for a description of this attribute.
671 Note: The effective set of policy intents for the service consists of any intents explicitly
672 stated in this requires attribute, combined with any intents specified for the service by the
673 implementation.
- 674 • **policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification](#)
675 [\[10\]](#) for a description of this attribute.

676

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

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

697

698 5.3 Reference

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

703

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

```

705 <!-- Component Reference schema snippet -->
706 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
707     ...
708     <component ... >*
709         <implementation ... />?
710         <service ... />*
711         <reference name="xs:NCName"
712             target="list of xs:anyURI"? autowire="xs:boolean"?
713             multiplicity="0..1 or 1..1 or 0..n or 1..n"?
714             wiredByImpl="xs:boolean"? requires="list of xs:QName"?
715             policySets="list of xs:QName"?>*
716         <interface ... />?
717         <binding uri="xs:anyURI"? requires="list of xs:QName"?
718             policySets="list of xs:QName"?/>*
719         <callback>?
720             <binding ... />+
721         </callback>
722     </reference>
723     <property ... />*
724 </component>
725     ...
726 </composite>
727

```

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

- 729 • **name : NCName (1..1)** – the name of the reference. Has to match a name of a reference
730 defined by the implementation.
- 731 • **autowire : boolean (0..1)** – whether the reference should be autowired, as described in
732 the [Autowire section](#). Default is false.
- 733 • **requires : QName (0..n)** – a list of policy intents. See the [Policy Framework specification](#)
734 [\[10\]](#) for a description of this attribute.
735 Note: The effective set of policy intents for the reference consists of any intents explicitly
736 stated in this requires attribute, combined with any intents specified for the reference by
737 the implementation.
- 738 • **policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification](#)
739 [\[10\]](#) for a description of this attribute.
- 740 • **multiplicity : 0..1/1..1/0..n/1..n (0..1)** - defines the number of wires that can connect
741 the reference to target services. Overrides the multiplicity specified for this reference on
742 the implementation. The value can only be equal or further restrict, i.e. 0..n to 0..1 or 1..n
743 to 1..1. The multiplicity can have the following values
 - 744 ○ 1..1 – one wire can have the reference as a source
 - 745 ○ 0..1 – zero or one wire can have the reference as a source
 - 746 ○ 1..n – one or more wires can have the reference as a source
 - 747 ○ 0..n - zero or more wires can have the reference as a source
- 748 • **target : anyURI (0..n)** – a list of one or more of target service URI's, depending on
749 multiplicity setting. Each value wires the reference to a component service that resolves

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

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

759

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

- 761 • **interface : Interface (0..1)** - A reference has **zero or one interface**, which describes
762 the operations required by the reference. The interface is described by an **interface**
763 **element** which is a child element of the reference element. If no interface is specified,
764 then the interface specified for the reference by the implementation is in effect. If an
765 interface is specified it must provide a compatible superset of the interface provided by the
766 implementation, i.e. provide a superset of the operations defined by the implementation
767 for the reference. For details on the interface element see [the Interface section](#).
- 768 • **binding : Binding (0..n)** - A reference element has **zero or more binding elements** as
769 children. If no bindings are specified, then the bindings specified for the reference by the
770 implementation are in effect. If any bindings are specified, then those bindings override
771 any and all the bindings specified by the implementation. Details of the binding element
772 are described in the [Bindings section](#). The binding, combined with any PolicySets in effect
773 for the binding, must satisfy the set of policy intents for the reference, as described in [the](#)
774 [Policy Framework specification \[10\]](#).

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

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

788

789 5.4 Property

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

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

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

800

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

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

805
806 So, for example, `source="$currency"` is used to reference a property of the composite
807 called "currency", while `source="$currency/a"` references the sub-part "a" of the
808 complex composite property with the name "currency".

809

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

812

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

815

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

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

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

822

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

824

```
825 <?xml version="1.0" encoding="UTF-8"?>
826 <!-- Component Property schema snippet -->
827 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
828   ...
829   <component ... >*
830     <implementation ... />?
831     <service ... />*
832     <reference ... />*
833     <property name="xs:NCName"
834       (type="xs:QName" | element="xs:QName")?
835       mustSupply="xs:boolean"? many="xs:boolean"?
836       source="xs:string"? file="xs:anyURI"?>*
837       property-value?
838     </property>
839   </component>
840   ...
841 </composite>
```

842

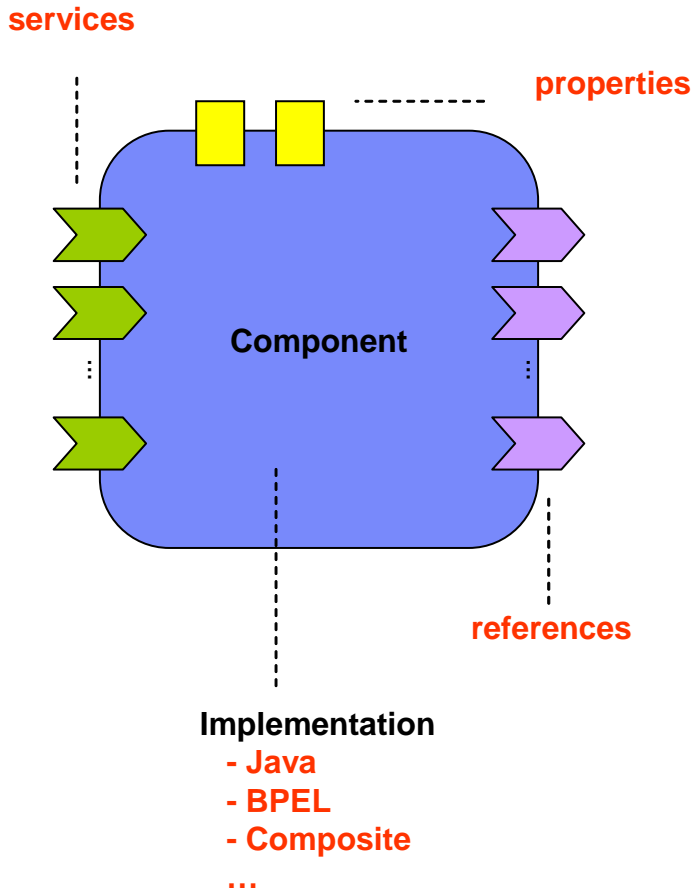
843 The **component property** element has the following **attributes**:

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

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

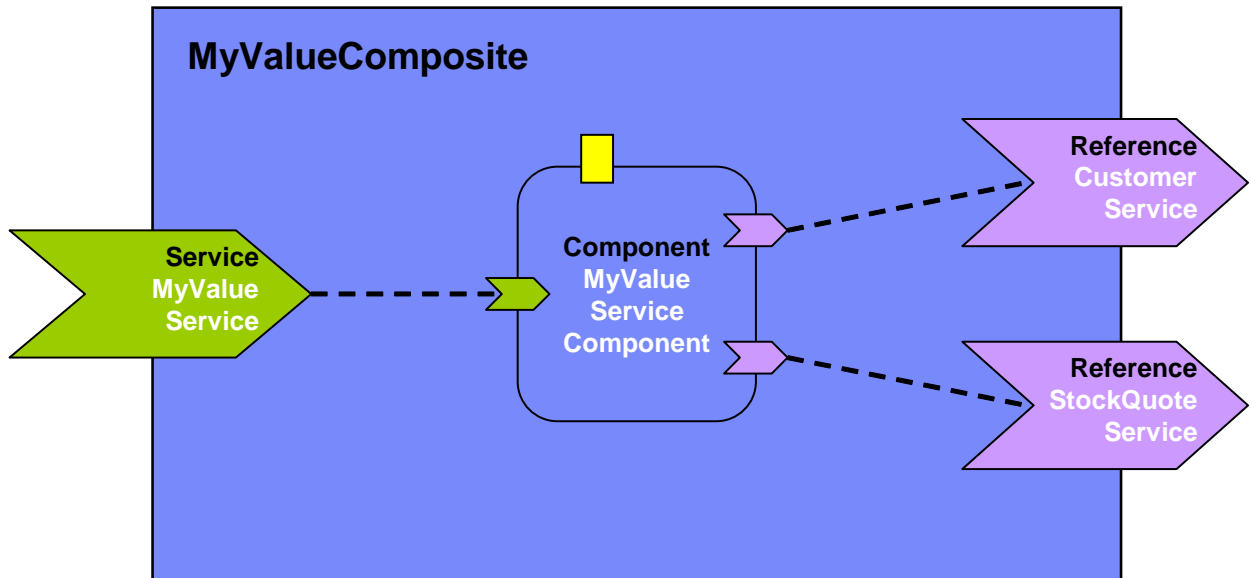
863 5.5 Example Component

864
865 The following figure shows the **component symbol** that is used to represent a component in an
866 assembly diagram.



867
868 *Figure 5: Component symbol*

869 The following figure shows the assembly diagram for the MyValueComposite containing the
870 MyValueServiceComponent.
871



872
873

874 *Figure 6: Assembly diagram for MyValueComposite*

875

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

880

```
881 <?xml version="1.0" encoding="ASCII"?>
882 <!-- MyValueComposite_1 example -->
883 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
884           targetNamespace="http://foo.com"
885           name="MyValueComposite" >
886
887     <service name="MyValueService" promote="MyValueServiceComponent"/>
888
889     <component name="MyValueServiceComponent">
890       <implementation.java
891 class="services.myvalue.MyValueServiceImpl"/>
892       <property name="currency">EURO</property>
893       <reference name="customerService"/>
894       <reference name="stockQuoteService"/>
895     </component>
896
897     <reference name="CustomerService"
898           promote="MyValueServiceComponent/customerService"/>
```

```

899
900     <reference name="StockQuoteService"
901         promote="MyValueServiceComponent/stockQuoteService"/>
902
903 </composite>

```

904
905 Note that the references of MyValueServiceComponent are explicitly declared only for purposes of
906 clarity – the references are defined by the MyValueServiceImpl implementation and there is no
907 need to redeclare them on the component unless the intention is to wire them or to override some
908 aspect of them.

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

```

912 <?xml version="1.0" encoding="ASCII"?>
913 <!-- MyValueComposite_2 example -->
914 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
915     targetNamespace="http://foo.com"
916     name="MyValueComposite" >
917
918     <service name="MyValueService" promote="MyValueServiceComponent"/>
919
920     <component name="MyValueServiceComponent">
921         <implementation.java
922 class="services.myvalue.MyValueServiceImpl"/>
923         <property name="currency">EURO</property>
924         <property name="currency">Yen</property>
925         <property name="currency">USDollar</property>
926         <reference name="customerService"
927             target="InternalCustomer/customerService"/>
928         <reference name="StockQuoteService"/>
929     </component>
930
931     ...
932
933     <reference name="CustomerService"
934         promote="MyValueServiceComponent/customerService"/>
935
936     <reference name="StockQuoteService"
937         promote="MyValueServiceComponent/StockQuoteService"/>
938
939 </composite>

```

940
941this assumes that the composite has another component called InternalCustomer (not shown)
942 which has a service to which the customerService reference of the MyValueServiceComponent is
943 wired as well as being promoted externally through the composite reference CustomerService.

944

945

946

6 Composite

947
948
949
950

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

951
952
953
954

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

955
956
957
958
959

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

960
961
962
963

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

964

965
966

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

967

968

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

969

```
<!-- Composite schema snippet -->
```

970

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

971

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

972

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

973

```
  autowire="xs:boolean"? constrainingType="QName"?
```

974

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

975

976

```
  <include ... />*
```

977

978

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

979

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

980

```
  <property ... />*
```

981

982

```
  <component ... />*
```

983

984

```
  <wire ... />*
```

985

986


```
</composite>
```

987

988


989

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

- 991 • **name** : **NCName (1..1)** – the name of the composite. The form of a composite name is
992 an XML QName, in the namespace identified by the targetNamespace attribute.
- 993 • **targetNamespace** : **anyURI (0..1)** – an identifier for a target namespace into which the
994 composite is declared
- 995 • **localExecutable**  **(0..1)** – whether all the components within the composite must all run in
996 the same operating system process. local="true" means that all the components must run
997 in the same process. local="false", which is the default, means that different components
998 within the composite may run in different operating system processes and they may even
999 run on different nodes on a network.
- 1000 • **autowire** : **boolean (0..1)** – whether contained component references should be
1001 autowired, as described in [the Autowire section](#). Default is false.
- 1002 • **constrainingType** : **QName (0..1)** – the name of a constrainingType. When specified,
1003 the set of services, references and properties of the composite, plus related intents, is
1004 constrained to the set defined by the constrainingType. See [the ConstrainingType Section](#)
1005 for more details.
- 1006 • **requires** : **QName (0..n)** – a list of policy intents. See the [Policy Framework](#)
1007 [specification \[10\]](#) for a description of this attribute.
- 1008 • **policySets** : **QName (0..n)** – a list of policy sets. See the [Policy Framework specification](#)
1009 [\[10\]](#) for a description of this attribute.

1010


1011 The **composite** element has the following **child elements**:


- 1012 • **service** : **CompositeService (0..n)** – see composite service section. 
- 1013 • **reference** : **CompositeReference (0..n)** – see composite reference section.
- 1014 • **property** : **CompositeProperty (0..n)** – see composite property section.
- 1015 • **component** : **Component (0..n)** – see component section.
- 1016 • **wire** : **Wire (0..n)** – see composite wire section.
- 1017 • **include** : **Include (0..n)** – see composite include section

1018

1019 Components contain configured implementations which hold the business logic of the composite.
1020 The components offer services and require references to other services. Composite services
1021 define the public services provided by the composite, which can be accessed from outside the
1022 composite. Composite references represent dependencies which the composite has on services
1023 provided elsewhere, outside the composite. Wires describe the connections between component
1024 services and component references within the composite. Included composites contribute the
1025 elements they contain to the using composite.

1026 Composite services involve the **promotion** of one service of one of the components within the
1027 composite, which means that the composite service is actually provided by one of the components
1028 within the composite. Composite references involve the **promotion** of one or more references of
1029 one or more components. Multiple component references can be promoted to the same composite
1030 reference, as long as all the component references are compatible with one another. Where
1031 multiple component references are promoted to the same composite reference, then they all share
1032 the same configuration, including the same target service(s).

1033 Composite services and composite references can use the configuration of their promoted services
1034 and references respectively (such as Bindings and Policy Sets). Alternatively composite services
1035 and composite references can override some or all of the configuration of the promoted services
1036 and references, through the configuration of bindings and other aspects of the composite service
1037 or reference. 

1038 Component services and component references can be promoted to composite services and
1039 references and also be wired internally within the composite at the same time. For a reference,
1040 this only makes sense if the reference supports a multiplicity greater than 1. 

1041

1042 6.1 Service

1043 The *services of a composite* are defined by promoting services defined by components
1044 contained in the composite. A component service is promoted by means of a composite *service*
1045 *element*.

1046 A composite service is represented by a *service element* which is a child of the composite
1047 element. There can be *zero or more* service elements in a composite. The following snippet
1048 shows the composite schema with the schema for a service child element:

1049

```
1050 <?xml version="1.0" encoding="ASCII"?>
1051 <!-- Composite Service schema snippet -->
1052 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
1053     ...
1054     <service name="xs:NCName" promote="xs:anyURI"
1055         requires="list of xs:QName"? policySets="list of xs:QName"?>*
1056         <interface ... />?
1057         <binding ... />*
1058         <callback?
1059             <binding ... />+
1060         </callback>
1061     </service>
1062     ...
1063 </composite>
```

1064

1065 The *composite service* element has the following *attributes*:

- 1066 • **name** : *NCName (1..1)* – the name of the service, the name MUST BE unique across all
1067 the composite services in the composite. The name of the composite service can be
1068 different from the name of the promoted component service.
- 1069 • **promote** : *anyURI (1..1)* – identifies the promoted service, the value is of the form
1070 <component-name>/<service-name>. The service name is optional if the target
1071 component only has one service. The same component service can be promoted by more
1072 than one composite service.
- 1073 • **requires** : *QName (0..n)* – a list of required policy intents. See the [Policy Framework](#)
1074 [specification \[10\]](#) for a description of this attribute. Specified *required intents* add to or
1075 further qualify the required intents defined by the promoted component service.
- 1076 • **policySets** : *QName (0..n)* – a list of policy sets. See the [Policy Framework specification](#)
1077 [\[10\]](#) for a description of this attribute.

1078

1079 The *composite service* element has the following *child elements*, whatever is not specified is
1080 defaulted from the promoted component service.

- 1081 • **interface** : *Interface (0..1)* - If an *interface* is specified it must be the same or a
1082 compatible subset of the interface provided by the promoted component service, i.e.
1083 provide a subset of the operations defined by the component service. The interface is

1084 described by **zero or one interface element** which is a child element of the service
 1085 element. For details on the interface element see [the Interface section](#).

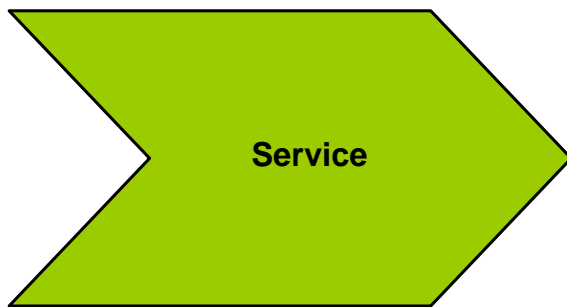
- 1086 • **binding : Binding (0..n)** - If bindings are specified they **override** the bindings defined
 1087 for the promoted component service from the composite service perspective. The bindings
 1088 defined on the component service are still in effect for local wires within the composite
 1089 that target the component service. A service element has zero or more **binding elements**
 1090 as children. Details of the binding element are described in the [Bindings section](#). For more
 1091 details on wiring see [the Wiring section](#).
- 1092 • **callback (0..1) / binding : Binding (1..n)** - A service element has an optional **callback**
 1093 element used if the interface has a callback defined,, which has one or more **binding**
 1094 elements as children. The **callback** and its binding child elements are specified if there is
 1095 a need to have binding details used to handle callbacks. If the callback element is not
 1096 present, the behaviour is runtime implementation dependent.

1097

1098 6.1.1 Service Examples

1099

1100 The following figure shows the service symbol that used to represent a service in an assembly
 1101 diagram:

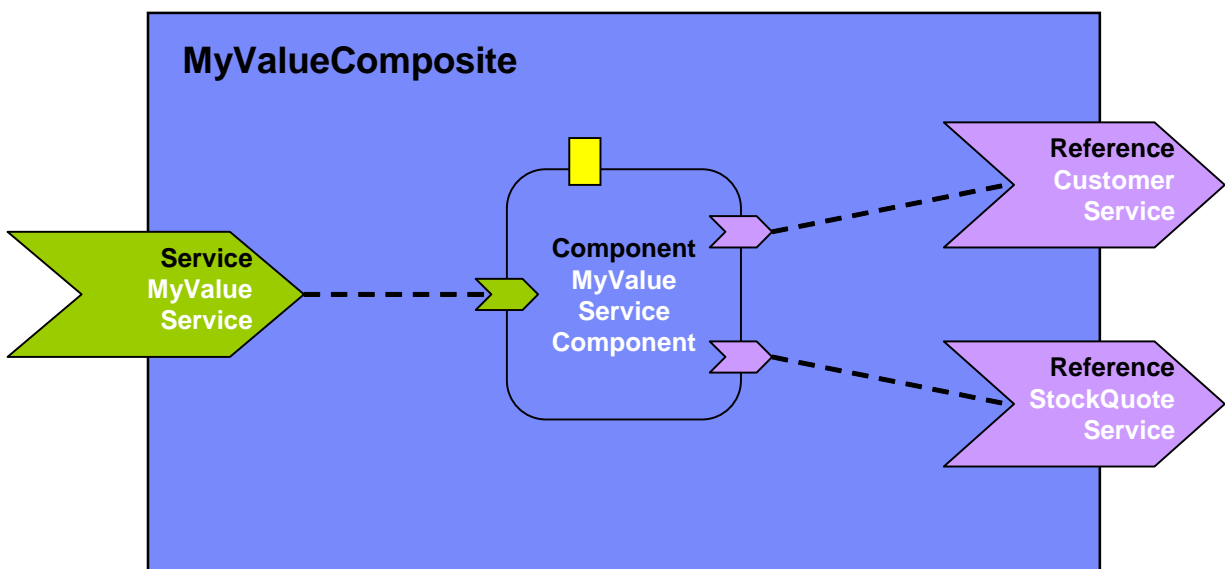


1102

1103 *Figure 7: Service symbol*

1104

1105 The following figure shows the assembly diagram for the MyValueComposite containing the service
 1106 MyValueService.



1107

1108 *Figure 8: MyValueComposite showing Service*

1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142

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

```
<?xml version="1.0" encoding="ASCII"?>
<!-- MyValueComposite_4 example -->
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
targetNamespace="http://foo.com"
name="MyValueComposite" >
...
<service name="MyValueService" promote="MyValueServiceComponent">
  <interface.java interface="services.myvalue.MyValueService"/>
  <binding.ws port="http://www.myvalue.org/MyValueService#
wsdl.endpoint(MyValueService/MyValueServiceSOAP)"/>
</service>
<component name="MyValueServiceComponent">
  <implementation.java
class="services.myvalue.MyValueServiceImpl"/>
  <property name="currency">EURO</property>
  <service name="MyValueService"/>
  <reference name="customerService"/>
  <reference name="StockQuoteService"/>
</component>
...
</composite>
```

6.2 Reference

1143
1144
1145
1146
1147
1148
1149
1150
1151
1152

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

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

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

```

1153 <!-- Composite Reference schema snippet -->
1154 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
1155   ...
1156   <reference name="xs:NCName" target="list of xs:anyURI"?
1157     promote="list of xs:anyURI" wiredByImpl="xs:boolean"?
1158     multiplicity="0..1 or 1..1 or 0..n or 1..n"?
1159     requires="list of xs:QName"? policySets="list of xs:QName"?>*
1160     <interface ... />?
1161     <binding ... />*
1162     <callback>?
1163       <binding ... />+
1164     </callback>
1165   </reference>
1166   ...
1167 </composite>

```

1170 The **composite reference** element has the following **attributes**:

- 1171 • **name : NCName (1..1)** – the name of the reference. The name must be unique across
1172 all the composite references in the composite. The name of the composite reference can
1173 be different then the name of the promoted component reference.
- 1174 • **promote : anyURI (1..n)** – identifies one or more promoted component references. The
1175 value is a list of values of the form <component-name>/<reference-name> separated by
1176 spaces. The specification of the reference name is optional if the component has only one
1177 reference.

1178 The same component reference maybe promoted more than once, using different
1179 composite references, but only if the multiplicity defined on the component reference is
1180 0..n or 1..n. The multiplicity on the composite reference can restrict accordingly.

1181 Two or more component references may be promoted by one composite reference, but
1182 only when

- 1183 • the interfaces of the component references are the same, or if the composite
1184 reference itself declares an interface then all the component references must have
1185 interfaces which are compatible with the composite reference interface
- 1186 • the multiplicities of the component references are compatible, i.e one can be the
1187 restricted form of the another, which also means that the composite reference
1188 carries the restricted form either implicitly or explicitly
- 1189 • the intents declared on the component references must be compatible – the
1190 intents which apply to the composite reference in this case are the union of the
1191 required intents specified for each of the promoted component references. If any
1192 intents contradict (eg mutually incompatible qualifiers for a particular intent) then
1193 there is an error.
- 1194 • **requires : QName (0..n)** – a list of required policy intents. See the [Policy Framework
1195 specification \[10\]](#) for a description of this attribute. Specified **required intents** add to or
1196 further qualify the required intents defined for the promoted component reference.
- 1197 • **policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification
1198 \[10\]](#) for a description of this attribute.
- 1199 • **multiplicity : 0..1|1..1|0..n|1..n (1..1)** - Defines the number of wires that can
1200 connect the reference to target services. The multiplicity can have the following values

- 1201 o 1..1 – one wire can have the reference as a source
- 1202 o 0..1 – zero or one wire can have the reference as a source
- 1203 o 1..n – one or more wires can have the reference as a source
- 1204 o 0..n - zero or more wires can have the reference as a source

1205 The value specified for the **multiplicity** attribute has to be compatible with the multiplicity
 1206 specified on the component reference, i.e. it has to be equal or further restrict. So a
 1207 composite reference of multiplicity 0..1 or 1..1 can be used where the promoted
 1208 component reference has multiplicity 0..n and 1..n respectively. However, a composite
 1209 reference of multiplicity 0..n or 1..n cannot be used to promote a component reference of
 1210 multiplicity 0..1 or 1..1 respectively.

- 1211 • **target : anyURI (0..n)** – a list of one or more of target service URI's, depending on
 1212 multiplicity setting. Each value wires the reference to a service in a composite that uses
 1213 the composite containing the reference as an implementation for one of its components. For
 1214 more details on wiring see [the section on Wires](#).
- 1215 • **wiredByImpl : boolean (0..1)** – a boolean value, "false" by default, which indicates that
 1216 the implementation wires this reference dynamically. If set to "true" it indicates that the
 1217 target of the reference is set at runtime by the implementation code (eg by the code
 1218 obtaining an endpoint reference by some means and setting this as the target of the
 1219 reference through the use of programming interfaces defined by the relevant Client and
 1220 Implementation specification). If "true" is set, then the reference should not be wired
 1221 statically within a using composite, but left unwired.

1222
 1223 The **composite reference** element has the following **child elements**, whatever is not specified is
 1224 defaulted from the promoted component reference(s).

- 1225 • **interface : Interface (0..1)** - If an **interface** is specified it must provide an interface
 1226 which is the same or which is a compatible superset of the interface declared by the
 1227 promoted component reference, i.e. provide a superset of the operations defined by the
 1228 component for the reference. The interface is described by **zero or one interface**
 1229 **element** which is a child element of the reference element. For details on the interface
 1230 element see [the Interface section](#).

- 1231 • **binding : Binding (0..n)** - If one or more **bindings** are specified they **override** any and
 1232 all of the bindings defined for the promoted component reference from the composite
 1233 reference perspective. The bindings defined on the component reference are still in effect
 1234 for local wires within the composite that have the component reference as their source. A
 1235 reference element has zero or more **binding elements** as children. Details of the binding
 1236 element are described in the [Bindings section](#). For more details on wiring see [the section](#)
 1237 [on Wires](#).

1238 Note that a binding element may specify an endpoint which is the target of that binding. A
 1239 reference must not mix the use of endpoints specified via binding elements with target
 1240 endpoints specified via the target attribute. If the target attribute is set, then binding
 1241 elements can only list one or more binding types that can be used for the wires identified
 1242 by the target attribute. All the binding types identified are available for use on each wire
 1243 in this case. If endpoints are specified in the binding elements, each endpoint must use
 1244 the binding type of the binding element in which it is defined. In addition, each binding
 1245 element needs to specify an endpoint in this case.

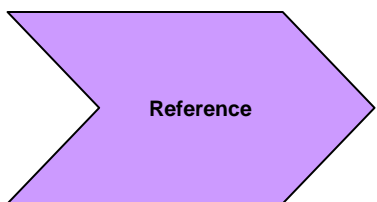
- 1246 • **callback (0..1) / binding : Binding (1..n)** - A **reference** element has an optional
 1247 **callback** element used if the interface has a callback defined, which has one or more
 1248 **binding** elements as children. The **callback** and its binding child elements are specified if
 1249 there is a need to have binding details used to handle callbacks. If the callback element is
 1250 not present, the behaviour is runtime implementation dependent.

1251

1252 6.2.1 Example Reference

1253

1254 The following figure shows the reference symbol that is used to represent a reference in an
1255 assembly diagram.



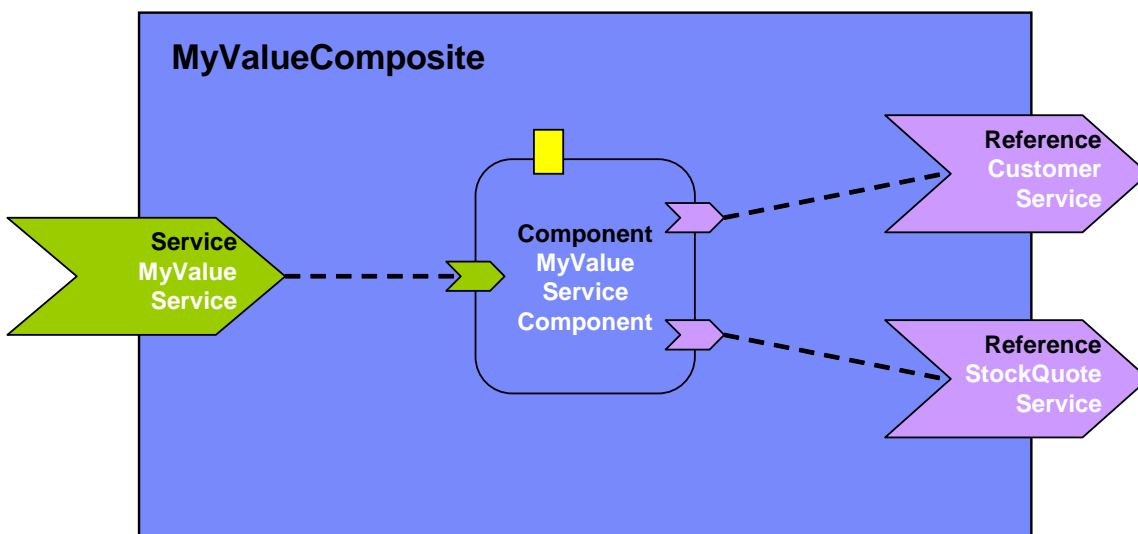
1256

1257 *Figure 9: Reference symbol*

1258

1259 The following figure shows the assembly diagram for the MyValueComposite containing the
1260 reference CustomerService and the reference StockQuoteService.

1261



1262

1263 *Figure 10: MyValueComposite showing References*

1264

1265 The following snippet shows the MyValueComposite.composite file for the MyValueComposite
1266 containing the reference elements for the CustomerService and the StockQuoteService. The
1267 reference CustomerService is bound using the SCA binding. The reference StockQuoteService is
1268 bound using the Web service binding. The endpoint addresses of the bindings can be specified,
1269 for example using the binding *uri* attribute (for details see the [Bindings](#) section), or overridden in an
1270 enclosing composite. Although in this case the reference StockQuoteService is bound to a Web
1271 service, its interface is defined by a Java interface, which was created from the WSDL portType of
1272 the target web service.

1273

```
1274 <?xml version="1.0" encoding="ASCII"?>  
1275 <!-- MyValueComposite_3 example -->  
1276 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
1277 targetNamespace="http://foo.com"  
1278 name="MyValueComposite" >
```

1279


```

1280     ...
1281
1282     <component name="MyValueServiceComponent">
1283         <implementation.java
1284 class="services.myvalue.MyValueServiceImpl"/>
1285         <property name="currency">EURO</property>
1286         <reference name="customerService"/>
1287         <reference name="StockQuoteService"/>
1288     </component>
1289
1290     <reference name="CustomerService"
1291         promote="MyValueServiceComponent/customerService">
1292         <interface.java interface="services.customer.CustomerService"/>
1293         <!-- The following forces the binding to be binding.sca whatever
1294 is -->
1295         <!-- specified by the component reference or by the underlying
1296 -->
1297         <!-- implementation
1298 -->
1299         <binding.sca/>
1300     </reference>
1301
1302     <reference name="StockQuoteService"
1303         promote="MyValueServiceComponent/StockQuoteService">
1304         <interface.java
1305 interface="services.stockquote.StockQuoteService"/>
1306         <binding.ws port="http://www.stockquote.org/StockQuoteService#
1307 wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
1308     </reference>
1309     ...
1310
1311     ...
1312
1313 </composite>
1314

```

1315 6.3 Property

1316 **Properties** allow for the configuration of an implementation with externally set data values. A
1317 composite can declare zero or more properties. Each property has a type, which may be either
1318 simple or complex. An implementation may also define a default value for a property. Properties
1319 are configured with values in the components that use the implementation.

1320 The declaration of a property in a composite follows the form described in the following schema
1321 snippet:

```

1322
1323 <?xml version="1.0" encoding="ASCII"?>
1324 <!-- Composite Property schema snippet -->

```

```

1325 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
1326   ...
1327   <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
1328     many="xs:boolean"? mustSupply="xs:boolean"?>*
1329     default-property-value?
1330   </property>
1331   ...
1332 </composite>
1333

```

1334 The **composite property** element has the following **attributes**:

- 1335 ▪ **name : NCName (1..1)** - the name of the property
- 1336 ▪ one of (1..1):
 - 1337 ○ **type : QName** – the type of the property - the qualified name of an XML schema
 - 1338 type
 - 1339 ○ **element : QName** – the type of the property defined as the qualified name of an
 - 1340 XML schema global element – the type is the type of the global element
- 1341 ▪ **many : boolean (0..1)** - whether the property is single-valued (false) or multi-valued
- 1342 (true). The default is **false**. In the case of a multi-valued property, it is presented to the
- 1343 implementation as a collection of property values.
- 1344 ▪ **mustSupply : boolean (0..1)** – whether the property value must be supplied by the
- 1345 component that uses the implementation – when mustSupply="true" the component must
- 1346 supply a value since the implementation has no default value for the property. A default-
- 1347 property-value should only be supplied when mustSupply="false" (the default setting for
- 1348 the mustSupply attribute), since the implication of a default value is that it is used only
- 1349 when a value is not supplied by the using component.

1350

1351 The property element may contain an optional **default-property-value**, which provides default

1352 value for the property. The default value must match the type declared for the property:

- 1353 ○ a string, if **type** is a simple type (must match the **type** declared)
- 1354 ○ a complex type value matching the type declared by **type**
- 1355 ○ an element matching the element named by **element**
- 1356 ○ multiple values are permitted if many="true" is specified

1357

1358 Implementation types other than **composite** can declare properties in an implementation-

1359 dependent form (eg annotations within a Java class), or through a property declaration of exactly

1360 the form described above in a componentType file.

1361 Property values can be configured when an implementation is used by a component. The form of

1362 the property configuration is shown in [the section on Components](#).

1363 6.3.1 Property Examples

1364

1365 For the following example of Property declaration and value setting, the following complex type is

1366 used as an example:

```

1367 <xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
1368   targetNamespace="http://foo.com/"
1369   xmlns:tns="http://foo.com/">

```

```

1370     <!-- ComplexProperty schema -->
1371     <xsd:element name="fooElement" type="MyComplexType"/>
1372     <xsd:complexType name="MyComplexType">
1373         <xsd:sequence>
1374             <xsd:element name="a" type="xsd:string"/>
1375             <xsd:element name="b" type="anyURI"/>
1376         </xsd:sequence>
1377         <attribute name="attr" type="xsd:string" use="optional"/>
1378     </xsd:complexType>
1379 </xsd:schema>
1380
1381 The following composite demonstrates the declaration of a property of a complex type, with a
1382 default value, plus it demonstrates the setting of a property value of a complex type within a
1383 component:
1384 <?xml version="1.0" encoding="ASCII"?>
1385
1386 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1387           xmlns:foo="http://foo.com"
1388           targetNamespace="http://foo.com"
1389           name="AccountServices">
1390 <!-- AccountServices Example1 -->
1391
1392     ...
1393
1394     <property name="complexFoo" type="foo:MyComplexType">
1395         <MyComplexPropertyValue xsi:type="foo:MyComplexType">
1396             <foo:a>AValue</foo:a>
1397             <foo:b>InterestingURI</foo:b>
1398         </MyComplexPropertyValue>
1399     </property>
1400
1401     <component name="AccountServiceComponent">
1402         <implementation.java class="foo.AccountServiceImpl"/>
1403         <property name="complexBar" source="$complexFoo"/>
1404         <reference name="accountDataService"
1405             target="AccountDataServiceComponent"/>
1406         <reference name="stockQuoteService" target="StockQuoteService"/>
1407     </component>
1408
1409     ...
1410
1411 </composite>

```

1412 In the declaration of the property named **complexFoo** in the composite **AccountServices**, the
1413 property is defined to be of type **foo:MyComplexType**. The namespace **foo** is declared in the

1414 composite and it references the example XSD, where MyComplexType is defined. The declaration
1415 of complexFoo contains a default value. This is declared as the content of the property element.
1416 In this example, the default value consists of the element *MyComplexPropertyValue* of type
1417 foo:MyComplexType and its two child elements <foo:a> and <foo:b>, following the definition of
1418 MyComplexType.

1419 In the component *AccountServiceComponent*, the component sets the value of the property
1420 *complexBar*, declared by the implementation configured by the component. In this case, the
1421 type of complexBar is foo:MyComplexType. The example shows that the value of the complexBar
1422 property is set from the value of the complexFoo property – the *source* attribute of the property
1423 element for complexBar declares that the value of the property is set from the value of a property
1424 of the containing composite. The value of the source attribute is *\$complexFoo*, where
1425 complexFoo is the name of a property of the composite. This value implies that the whole of the
1426 value of the source property is used to set the value of the component property.

1427 The following example illustrates the setting of the value of a property of a simple type (a string)
1428 from *part* of the value of a property of the containing composite which has a complex type:

```
1429 <?xml version="1.0" encoding="ASCII"?>
1430
1431 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1432           xmlns:foo="http://foo.com"
1433           targetNamespace="http://foo.com"
1434           name="AccountServices">
1435 <!-- AccountServices Example2 -->
1436
1437     ...
1438
1439     <property name="complexFoo" type="foo:MyComplexType">
1440         <MyComplexPropertyValue xsi:type="foo:MyComplexType">
1441             <foo:a>AValue</foo:a>
1442             <foo:b>InterestingURI</foo:b>
1443         </MyComplexPropertyValue>
1444     </property>
1445
1446     <component name="AccountServiceComponent">
1447         <implementation.java class="foo.AccountServiceImpl"/>
1448         <property name="currency" source="$complexFoo/a"/>
1449         <reference name="accountDataService"
1450             target="AccountDataServiceComponent"/>
1451         <reference name="stockQuoteService" target="StockQuoteService"/>
1452     </component>
1453
1454     ...
1455
1456 </composite>
```

1457 In this example, the component *AccountServiceComponent* sets the value of a property called
1458 *currency*, which is of type string. The value is set from a property of the composite
1459 *AccountServices* using the source attribute set to *\$complexFoo/a*. This is an XPath expression

1460 that selects the property name *complexFoo* and then selects the value of the *a* subelement of
1461 complexFoo. The "a" subelement is a string, matching the type of the currency property.

1462 Further examples of declaring properties and setting property values in a component follow:

1463 Declaration of a property with a simple type and a default value:

```
1464 <property name="SimpleTypeProperty" type="xsd:string">  
1465 MyValue  
1466 </property>
```

1467

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

```
1469 <property name="complexFoo" type="foo:MyComplexType">  
1470   <MyComplexPropertyValue xsi:type="foo:MyComplexType">  
1471     <foo:a>AValue</foo:a>  
1472     <foo:b>InterestingURI</foo:b>  
1473   </MyComplexPropertyValue>  
1474 </property>
```

1475

1476 Declaration of a property with an element type:

```
1477 <property name="elementFoo" element="foo:fooElement">  
1478   <foo:fooElement>  
1479     <foo:a>AValue</foo:a>  
1480     <foo:b>InterestingURI</foo:b>  
1481   </foo:fooElement>  
1482 </property>
```

1483

1484 Property value for a simple type:

```
1485 <property name="SimpleTypeProperty">  
1486 MyValue  
1487 </property>
```

1488

1489

1490 Property value for a complex type, also showing the setting of an attribute value of the complex
1491 type:

```
1492 <property name="complexFoo">  
1493   <MyComplexPropertyValue xsi:type="foo:MyComplexType" attr="bar">  
1494     <foo:a>AValue</foo:a>  
1495     <foo:b>InterestingURI</foo:b>  
1496   </MyComplexPropertyValue>  
1497 </property>
```

1498

1499 Property value for an element type:

```
1500 <property name="elementFoo">  
1501   <foo:fooElement attr="bar">  
1502     <foo:a>AValue</foo:a>
```

```
1503     <foo:b>InterestingURI</foo:b>
1504   </foo:fooElement>
1505 </property>
```

1506

1507 Declaration of a property with a complex type where multiple values are supported:

```
1508 <property name="complexFoo" type="foo:MyComplexType" many="true"/>
1509
```

1510 Setting of a value for that property where multiple values are supplied:

```
1511 <property name="complexFoo">
1512   <MyComplexPropertyValue1 xsi:type="foo:MyComplexType" attr="bar">
1513     <foo:a>AValue</foo:a>
1514     <foo:b>InterestingURI</foo:b>
1515   </MyComplexPropertyValue1>
1516   <MyComplexPropertyValue2 xsi:type="foo:MyComplexType" attr="zing">
1517     <foo:a>BValue</foo:a>
1518     <foo:b>BoringURI</foo:b>
1519   </MyComplexPropertyValue2>
1520 </property>
1521
```

1522 6.4 Wire

1523 **SCA wires** within a composite connect *source component references* to *target component*
1524 *services*.

1525 One way of defining a wire is by *configuring a reference of a component using its target*
1526 *attribute*. The reference element is configured with the wire-target-URI of the service(s) that
1527 resolve the reference. Multiple target services are valid when the reference has a multiplicity of
1528 0..n or 1..n.

1529 An alternative way of defining a Wire is by means of a *wire element* which is a child of the
1530 composite element. There can be *zero or more* wire elements in a composite. This alternative
1531 method for defining wires is useful in circumstances where separation of the wiring from the
1532 elements the wires connect helps simplify development or operational activities. An example is
1533 where the components used to build a domain are relatively static but where new or changed
1534 applications are created regularly from those components, through the creation of new assemblies
1535 with different wiring. Deploying the wiring separately from the components allows the wiring to
1536 be created or modified with minimum effort.

1537 Note that a Wire specified via a wire element is equivalent to a wire specified via the target
1538 attribute of a reference. The rule which forbids mixing of wires specified with the target attribute
1539 with the specification of endpoints in binding subelements of the reference also applies to wires
1540 specified via separate wire elements.

1541 The following snippet shows the composite schema with the schema for the reference elements of
1542 components and composite services and the wire child element:

1543

```
1544 <?xml version="1.0" encoding="ASCII"?>
1545 <!-- Wires schema snippet -->
1546 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1547           targetNamespace="xs:anyURI"
1548           name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?
```

```

1549         constrainingType="QName"?
1550         requires="list of xs:QName"? policySets="list of
1551 xs:QName"?>
1552
1553     ...
1554
1555     <wire source="xs:anyURI" target="xs:anyURI" />*
1556
1557 </composite>

```

1560 The **reference element of a component** and the **reference element of a service** has a list of
1561 one or more of the following **wire-target-URI** values for the target, with multiple values
1562 separated by a space:

- 1563 • <component-name>/<service-name>
 - 1564 ○ where the target is a service of a component. The specification of the service
 - 1565 name is optional if the target component only has one service with a compatible
 - 1566 interface

1567

1568 The **wire element** has the following attributes:

- 1569 • **source (required)** – names the source component reference. Valid URI schemes are:
 - 1570 ○ <component-name>/<reference-name>
 - 1571 ▪ where the source is a component reference. The specification of the
 - 1572 reference name is optional if the source component only has one reference
- 1573 • **target (required)** – names the target component service. Valid URI schemes are
 - 1574 ○ <component-name>/<service-name>
 - 1575 ▪ where the target is a service of a component. The specification of the
 - 1576 service name is optional if the target component only has one service with
 - 1577 a compatible interface

1578 For a composite used as a component implementation, wires can only link sources and targets
1579 that are contained in the same composite (irrespective of which file or files are used to describe
1580 the composite). Wiring to entities outside the composite is done through services and references
1581 of the composite with wiring defined by the next higher composite.

1582 A wire may only connect a source to a target if the target implements an interface that is
1583 compatible with the interface required by the source. The source and the target are compatible if:

- 1584 1. the source interface and the target interface MUST either both be remotable or they are
1585 both local
- 1586 2. the operations on the target interface MUST be the same as or be a superset of the
1587 operations in the interface specified on the source
- 1588 3. compatibility for the individual operation is defined as compatibility of the signature, that
1589 is operation name, input types, and output types MUST BE the same.
- 1590 4. the order of the input and output types also MUST BE the same.
- 1591 5. the set of Faults and Exceptions expected by the source MUST BE the same or be a
1592 superset of those specified by the target.
- 1593 6. other specified attributes of the two interfaces MUST match, including Scope and Callback
1594 interface

1595 A Wire can connect between different interface languages (eg. Java interfaces and WSDL
1596 portTypes) in either direction, as long as the operations defined by the two interface types are
1597 equivalent. They are equivalent if the operation(s), parameter(s), return value(s) and
1598 faults/exceptions map to each other.

1599 Service clients cannot (portably) ask questions at runtime about additional interfaces that are
1600 provided by the implementation of the service (e.g. the result of "instance of" in Java is non
1601 portable). It is valid for an SCA implementation to have proxies for all wires, so that, for example,
1602 a reference object passed to an implementation may only have the business interface of the
1603 reference and may not be an instance of the (Java) class which is used to implement the target
1604 service, even where the interface is local and the target service is running in the same process.

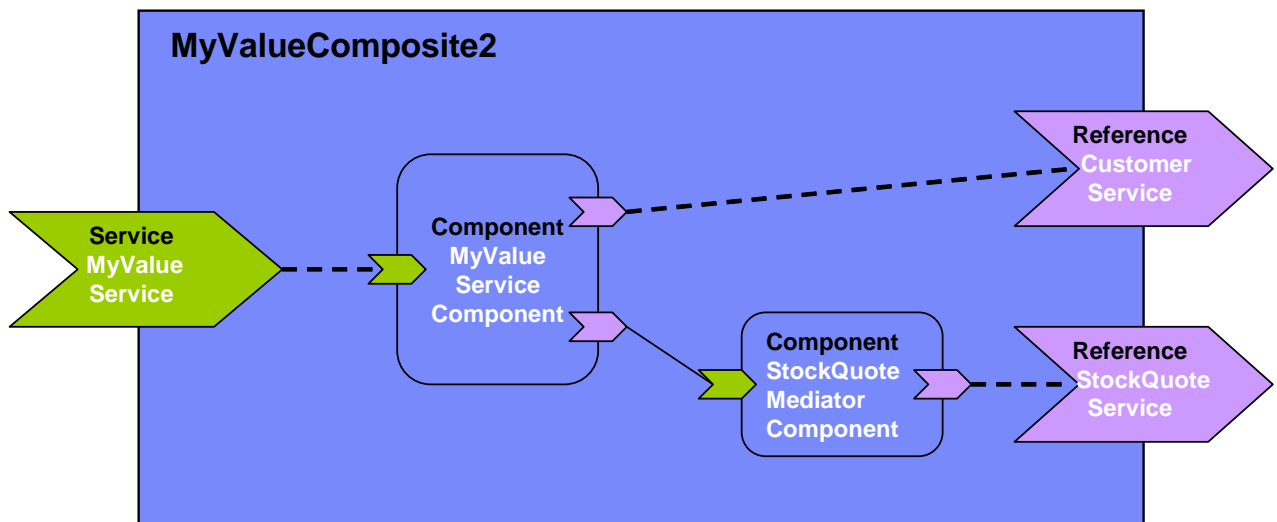
1605 **Note:** It is permitted to deploy a composite that has references that are not wired. For the case of
1606 an un-wired reference with multiplicity 1..1 or 1..n the deployment process provided by an SCA
1607 runtime SHOULD issue a warning.

1608

1609 6.4.1 Wire Examples

1610

1611 The following figure shows the assembly diagram for the MyValueComposite2 containing wires
1612 between service, components and references.



1613

1614 *Figure 11: MyValueComposite2 showing Wires*

1615

1616 The following snippet shows the MyValueComposite2.composite file for the MyValueComposite2
1617 containing the configured component and service references. The service MyValueService is wired
1618 to the MyValueServiceComponent. The MyValueServiceComponent's customerService reference is
1619 wired to the composite's CustomerService reference. The MyValueServiceComponent's
1620 stockQuoteService reference is wired to the StockQuoteMediatorComponent, which in turn has its
1621 reference wired to the StockQuoteService reference of the composite.

1622

```
1623 <?xml version="1.0" encoding="ASCII"?>  
1624 <!-- MyValueComposite Wires examples -->  
1625 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
1626 targetNamespace="http://foo.com"  
1627 name="MyValueComposite2" >
```

1628


```

1629     <service name="MyValueService" promote="MyValueServiceComponent">
1630         <interface.java interface="services.myvalue.MyValueService"/>
1631         <binding.ws port="http://www.myvalue.org/MyValueService#
1632             wsdl.endpoint(MyValueService/MyValueServiceSOAP)"/>
1633     </service>
1634
1635     <component name="MyValueServiceComponent">
1636         <implementation.java
1637 class="services.myvalue.MyValueServiceImpl"/>
1638         <property name="currency">EURO</property>
1639         <service name="MyValueService"/>
1640         <reference name="customerService"/>
1641         <reference name="stockQuoteService"
1642             target="StockQuoteMediatorComponent"/>
1643     </component>
1644
1645     <component name="StockQuoteMediatorComponent">
1646         <implementation.java class="services.myvalue.SQMediatorImpl"/>
1647         <property name="currency">EURO</property>
1648         <reference name="stockQuoteService"/>
1649     </component>
1650
1651     <reference name="CustomerService"
1652         promote="MyValueServiceComponent/customerService">
1653         <interface.java interface="services.customer.CustomerService"/>
1654         <binding.sca/>
1655     </reference>
1656
1657     <reference name="StockQuoteService"
1658 promote="StockQuoteMediatorComponent">
1659         <interface.java
1660 interface="services.stockquote.StockQuoteService"/>
1661         <binding.ws port="http://www.stockquote.org/StockQuoteService#
1662             wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
1663     </reference>
1664
1665 </composite>
1666

```

1667 6.4.2 Autowire

1668 SCA provides a feature named **Autowire**, which can help to simplify the assembly of composites.
1669 Autowire enables component references to be automatically wired to component services which
1670 will satisfy those references, without the need to create explicit wires between the references and
1671 the services. When the autowire feature is used, a component reference which is not promoted
1672 and which is not explicitly wired to a service within a composite is automatically wired to a target

1673 service within the same composite. Autowire works by searching within the composite for a
1674 service interface which matches the interface of the references.

1675 The autowire feature is not used by default. Autowire is enabled by the setting of an autowire
1676 attribute to "true". Autowire is disabled by setting of the autowire attribute to "false" The autowire
1677 attribute can be applied to any of the following elements within a composite:

- 1678 • reference
- 1679 • component
- 1680 • composite

1681 Where an element does not have an explicit setting for the autowire attribute, it inherits the
1682 setting from its parent element. Thus a reference element inherits the setting from its containing
1683 component. A component element inherits the setting from its containing composite. Where
1684 there is no setting on any level, autowire="false" is the default.

1685 As an example, if a composite element has autowire="true" set, this means that autowiring is
1686 enabled for all component references within that composite. In this example, autowiring can be
1687 turned off for specific components and specific references through setting autowire="false" on the
1688 components and references concerned.

1689 For each component reference for which autowire is enabled, the autowire process searches within
1690 the composite for target services which are compatible with the reference. "Compatible" here
1691 means:

- 1692 • the target service interface must be a compatible superset of the reference interface (as
1693 defined in [the section on Wires](#))
- 1694 • the intents, bindings and policies applied to the service must be compatible on the
1695 reference – so that wiring the reference to the service will not cause an error due to
1696 binding and policy mismatch (see [the Policy Framework specification \[10\]](#) for details)

1697 If the search finds **more than 1** valid target service for a particular reference, the action taken
1698 depends on the multiplicity of the reference:

- 1699 • for multiplicity 0..1 and 1..1, the SCA runtime selects one of the target services in a
1700 runtime-dependent fashion and wires the reference to that target service
- 1701 • for multiplicity 0..n and 1..n, the reference is wired to all of the target services

1702 If the search finds **no** valid target services for a particular reference, the action taken depends on
1703 the multiplicity of the reference:

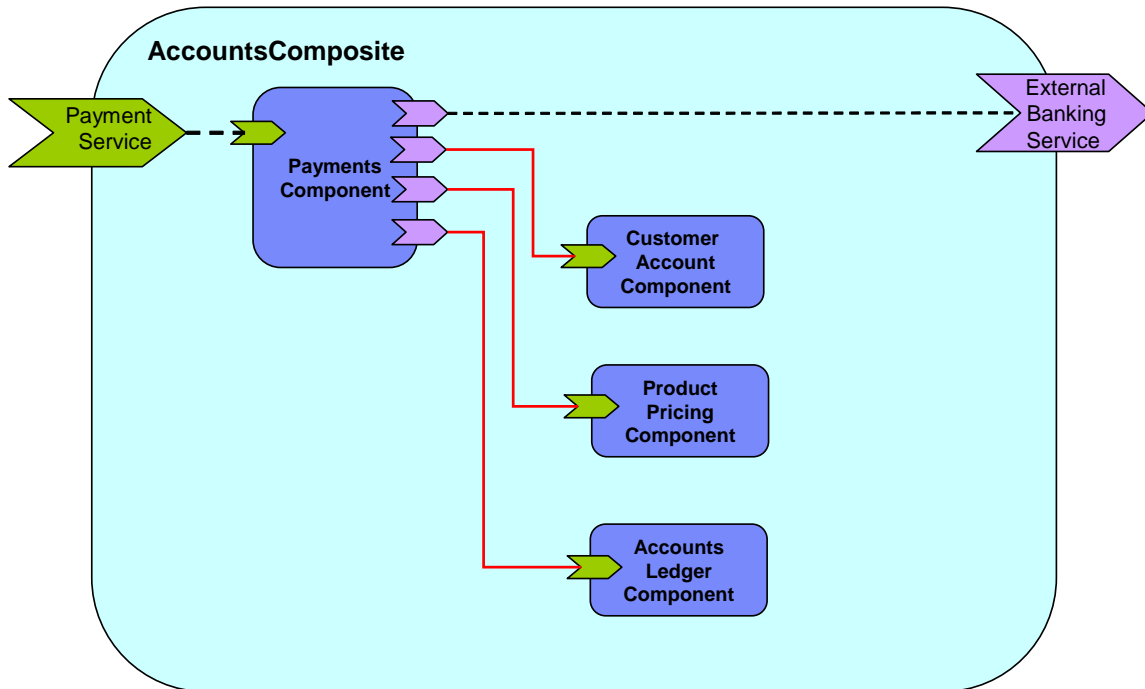
- 1704 • for multiplicity 0..1 and 0..n, there is no problem – no services are wired and there is no
1705 error
- 1706 • for multiplicity 1..1 and 1..n, an error is raised by the SCA runtime since the reference is
1707 intended to be wired

1708

1709 6.4.3 Autowire Examples

1710 This example demonstrates two versions of the same composite – the first version is done using
1711 explicit wires, with no autowiring used, the second version is done using autowire. In both cases
1712 the end result is the same – the same wires connect the references to the services.

1713 First, here is a diagram for the composite:



1714
 1715 *Figure 12: Example Composite for Autowire*
 1716 First, the composite using explicit wires:

```

1717 <?xml version="1.0" encoding="UTF-8"?>
1718 <!-- Autowire Example - No autowire -->
1719 <composite xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
1720           xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1721           xmlns:foo="http://foo.com"
1722           targetNamespace="http://foo.com"
1723           name="AccountComposite">
1724
1725   <service name="PaymentService" promote="PaymentsComponent" />
1726
1727   <component name="PaymentsComponent">
1728     <implementation.java class="com.foo.accounts.Payments" />
1729     <service name="PaymentService" />
1730     <reference name="CustomerAccountService"
1731               target="CustomerAccountComponent" />
1732     <reference name="ProductPricingService"
1733               target="ProductPricingComponent" />
1734     <reference name="AccountsLedgerService"
1735               target="AccountsLedgerComponent" />
1736     <reference name="ExternalBankingService" />
1737   </component>
1738
1739   <component name="CustomerAccountComponent">

```

```

1740         <implementation.java class="com.foo.accounts.CustomerAccount" />
1741     </component>
1742
1743     <component name="ProductPricingComponent">
1744         <implementation.java class="com.foo.accounts.ProductPricing" />
1745     </component>
1746
1747     <component name="AccountsLedgerComponent">
1748         <implementation.composite name="foo:AccountsLedgerComposite" />
1749     </component>
1750
1751     <reference name="ExternalBankingService"
1752         promote="PaymentsComponent/ExternalBankingService" />
1753
1754 </composite>
1755

```

1756 Secondly, the composite using autowire:

```

1757 <?xml version="1.0" encoding="UTF-8"?>
1758 <!-- Autowire Example - With autowire -->
1759 <composite xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
1760     xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1761     xmlns:foo="http://foo.com"
1762     targetNamespace="http://foo.com"
1763     name="AccountComposite">
1764
1765     <service name="PaymentService" promote="PaymentsComponent">
1766         <interface.java class="com.foo.PaymentServiceInterface" />
1767     </service>
1768
1769     <component name="PaymentsComponent" autowire="true">
1770         <implementation.java class="com.foo.accounts.Payments" />
1771         <service name="PaymentService" />
1772         <reference name="CustomerAccountService" />
1773         <reference name="ProductPricingService" />
1774         <reference name="AccountsLedgerService" />
1775         <reference name="ExternalBankingService" />
1776     </component>
1777
1778     <component name="CustomerAccountComponent">
1779         <implementation.java class="com.foo.accounts.CustomerAccount" />
1780     </component>
1781
1782     <component name="ProductPricingComponent">

```

```

1783     <implementation.java class="com.foo.accounts.ProductPricing"/>
1784 </component>
1785
1786 <component name="AccountsLedgerComponent">
1787     <implementation.composite name="foo:AccountsLedgerComposite"/>
1788 </component>
1789
1790 <reference name="ExternalBankingService"
1791     promote="PaymentsComponent/ExternalBankingService"/>
1792
1793 </composite>

```

1794 In this second case, autowire is set on for the PaymentsComponent and there are no explicit wires
1795 for any of its references – the wires are created automatically through autowire.

1796 **Note:** In the second example, it would be possible to omit all of the service and reference
1797 elements from the PaymentsComponent. They are left in for clarity, but if they are omitted, the
1798 component service and references still exist, since they are provided by the implementation used
1799 by the component.

1800

1801 6.5 Using Composites as Component Implementations

1802 Composites may form *component implementations* in higher-level composites – in other words
1803 the higher-level composites can have components which are implemented by composites.

1804 When a composite is used as a component implementation, it defines a boundary of visibility.
1805 Components within the composite cannot be referenced directly by the using component. The
1806 using component can only connect wires to the services and references of the used composite and
1807 set values for any properties of the composite. The internal construction of the composite is
1808 invisible to the using component.

1809 A composite used as a component implementation must also honor a *completeness contract*.
1810 The services, references and properties of the composite form a contract which is relied upon by
1811 the using component. The concept of completeness of the composite implies:

- 1812 • the composite must have at least one service or at least one reference.
1813 A component with no services and no references is not meaningful in terms of SCA, since
1814 it cannot be wired to anything – it neither provides nor consumes any services
1815
- 1816 • each service offered by the composite must be wired to a service of a component or to a
1817 composite reference.
1818 If services are left unwired, the implication is that some exception will occur at runtime if
1819 the service is invoked.

1820 The component type of a composite is defined by the set of service elements, reference elements
1821 and property elements that are the children of the composite element.

1822 Composites are used as component implementations through the use of the
1823 *implementation.composite* element as a child element of the component. The schema snippet
1824 for the implementation.composite element is:

```

1825
1826 <?xml version="1.0" encoding="ASCII"?>
1827 <!-- Composite Implementation schema snippet -->
1828 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1829     targetNamespace="xs:anyURI"

```

```

1830         name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?
1831         constrainingType="QName"?
1832         requires="list of xs:QName"? policySets="list of
1833 xs:QName"?>
1834
1835     ...
1836
1837     <component name="xs:NCName" autowire="xs:boolean"?
1838         requires="list of xs:QName"? policySets="list of xs:QName"?>*
1839         <implementation.composite name="xs:QName"/>?
1840         <service name="xs:NCName" requires="list of xs:QName"?
1841             policySets="list of xs:QName"?>*
1842             <interface ... />?
1843             <binding uri="xs:anyURI" name="xs:QName"?
1844                 requires="list of xs:QName"
1845                 policySets="list of xs:QName"?/>*
1846             <callback?
1847                 <binding uri="xs:anyURI"? name="xs:QName"?
1848                     requires="list of xs:QName"?
1849                     policySets="list of xs:QName"?/>+
1850             </callback>
1851         </service>
1852         <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
1853             source="xs:string"? file="xs:anyURI"?>*
1854             property-value
1855         </property>
1856         <reference name="xs:NCName" target="list of xs:anyURI"?
1857             autowire="xs:boolean"? wiredByImpl="xs:boolean"?
1858             requires="list of xs:QName"? policySets="list of xs:QName"?
1859             multiplicity="0..1 or 1..1 or 0..n or 1..n"?/>*
1860             <interface ... />?
1861             <binding uri="xs:anyURI"? name="xs:QName"?
1862                 requires="list of xs:QName" policySets="list of
1863 xs:QName"?/>*
1864             <callback?
1865                 <binding uri="xs:anyURI"? name="xs:QName"?
1866                     requires="list of xs:QName"?
1867                     policySets="list of xs:QName"?/>+
1868             </callback>
1869             </reference>
1870         </component>
1871
1872     ...

```

1873
1874 `</composite>`

1875
1876
1877 The implementation.composite element has the following attribute:

- 1878 • **name (required)** – the name of the composite used as an implementation
- 1879

1880 6.5.1 Example of Composite used as a Component Implementation

1881
1882 The following is an example of a composite which contains two components, each of which is
1883 implemented by a composite:
1884

```
1885 <?xml version="1.0" encoding="UTF-8"?>
1886 <!-- CompositeComponent example -->
1887 <composite xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
1888     xsd:schemaLocation="http://docs.oasis-open.org/ns/opencsa/sca/200712
1889     file:/C:/Strategy/SCA/v09_osoaschemas/schemas/sca.xsd"
1890     xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1891     targetNamespace="http://foo.com"
1892     xmlns:foo="http://foo.com"
1893     name="AccountComposite">
1894
1895     <service name="AccountService" promote="AccountServiceComponent">
1896         <interface.java interface="services.account.AccountService"/>
1897         <binding.ws port="AccountService#
1898             wsdl.endpoint(AccountService/AccountServiceSOAP)"/>
1899     </service>
1900
1901     <reference name="stockQuoteService"
1902         promote="AccountServiceComponent/StockQuoteService">
1903         <interface.java
1904     interface="services.stockquote.StockQuoteService"/>
1905         <binding.ws
1906     port="http://www.quickstockquote.com/StockQuoteService#
1907             wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
1908     </reference>
1909
1910     <property name="currency" type="xsd:string">EURO</property>
1911
1912     <component name="AccountServiceComponent">
1913         <implementation.composite name="foo:AccountServiceCompositel"/>
1914
1915     <reference name="AccountDataService" target="AccountDataService"/>
```

```

1916         <reference name="StockQuoteService"/>
1917
1918         <property name="currency" source="$currency"/>
1919     </component>
1920
1921     <component name="AccountDataService">
1922         <implementation.composite name="foo:AccountDataServiceComposite"/>
1923
1924         <property name="currency" source="$currency"/>
1925     </component>
1926
1927 </composite>
1928

```

1929 6.6 Using Composites through Inclusion

1930 In order to assist team development, composites may be developed in the form of multiple
1931 physical artifacts that are merged into a single logical unit.

1932 A composite is defined in an *xxx.composite* file and the composite may receive additional
1933 content through the *inclusion of other composite* files.

1934 The semantics of included composites are that the content of the included composite is inlined into
1935 the using composite *xxx.composite* file through *include* elements in the using composite. The
1936 effect is one of *textual inclusion* – that is, the text content of the included composite is placed
1937 into the using composite in place of the include statement. The included composite element itself
1938 is discarded in this process – only its contents are included.

1939 The composite file used for inclusion can have any contents, but always contains a single
1940 *composite* element. The composite element may contain any of the elements which are valid as
1941 child elements of a composite element, namely components, services, references, wires and
1942 includes. There is no need for the content of an included composite to be complete, so that
1943 artifacts defined within the using composite or in another associated included composite file may
1944 be referenced. For example, it is permissible to have two components in one composite file while a
1945 wire specifying one component as the source and the other as the target can be defined in a
1946 second included composite file.

1947 It is an error if the (using) composite resulting from the inclusion is invalid – for example, if there
1948 are duplicated elements in the using composite (eg. two services with the same uri contributed by
1949 different included composites), or if there are wires with non-existent source or target.

1950 The following snippet shows the partial schema for the include element.

```

1951
1952 <?xml version="1.0" encoding="UTF-8"?>
1953 <!-- Include snippet -->
1954 <composite      xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1955                targetNamespace="xs:anyURI"
1956                name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?
1957                constrainingType="QName"?
1958                requires="list of xs:QName"? policySets="list of
1959 xs:QName"?>
1960
1961     ...

```



```

1962
1963     <include name="xs:QName" />*
1964
1965     ...
1966
1967 </composite>
1968

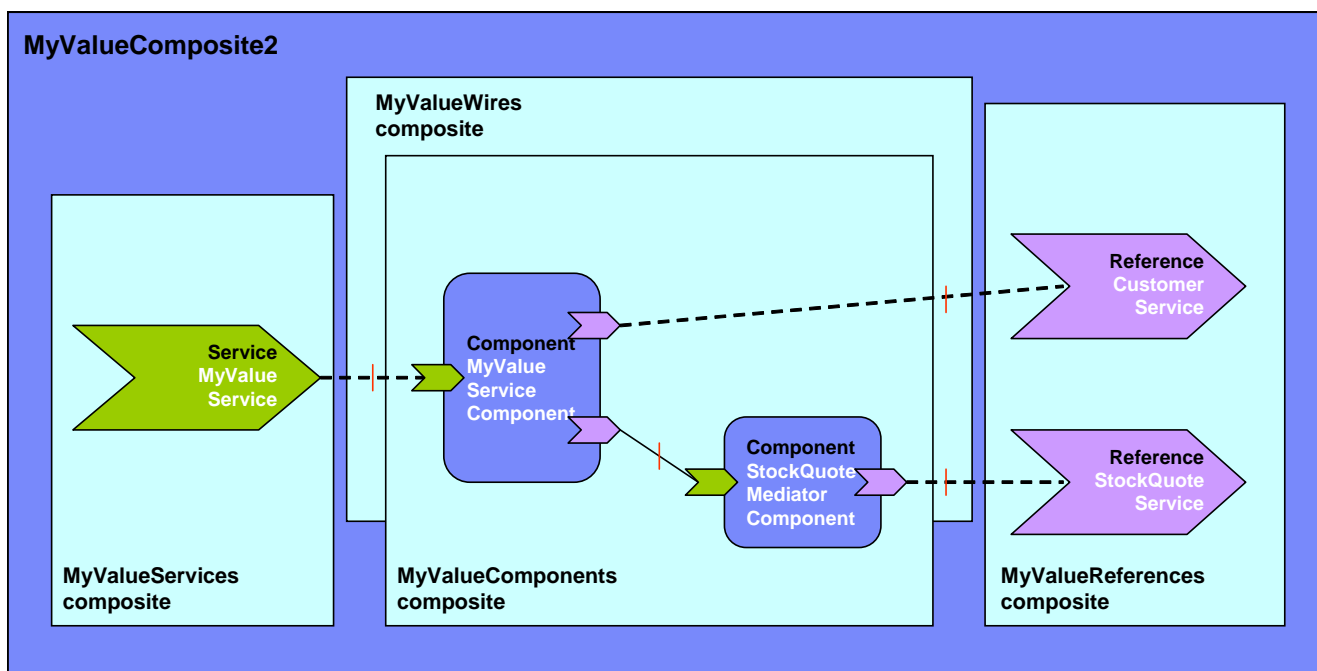
```

1969 The include element has the following *attribute*:

- 1970 • **name (required)** – the name of the composite that is included.

1972 6.6.1 Included Composite Examples

1973
1974 The following figure shows the assembly diagram for the MyValueComposite2 containing four
1975 included composites. The **MyValueServices composite** contains the MyValueService service. The
1976 **MyValueComponents composite** contains the MyValueServiceComponent and the
1977 StockQuoteMediatorComponent as well as the wire between them. The **MyValueReferences**
1978 **composite** contains the CustomerService and StockQuoteService references. The **MyValueWires**
1979 **composite** contains the wires that connect the MyValueService service to the
1980 MyValueServiceComponent, that connect the customerService reference of the
1981 MyValueServiceComponent to the CustomerService reference, and that connect the
1982 stockQuoteService reference of the StockQuoteMediatorComponent to the StockQuoteService
1983 reference. Note that this is just one possible way of building the MyValueComposite2 from a set of
1984 included composites.



1985
1986
1987 *Figure 13 MyValueComposite2 built from 4 included composites*

1988
1989 The following snippet shows the contents of the MyValueComposite2.composite file for the
1990 MyValueComposite2 built using included composites. In this sample it only provides the name of

1991 the composite. The composite file itself could be used in a scenario using included composites to
1992 define components, services, references and wires.

1993

```
1994 <?xml version="1.0" encoding="ASCII"?>
1995 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1996           targetNamespace="http://foo.com"
1997           xmlns:foo="http://foo.com"
1998           name="MyValueComposite2" >
1999
2000     <include name="foo:MyValueServices" />
2001     <include name="foo:MyValueComponents" />
2002     <include name="foo:MyValueReferences" />
2003     <include name="foo:MyValueWires" />
2004
2005 </composite>
```

2006
2007 The following snippet shows the content of the MyValueServices.composite file.

2008

```
2009 <?xml version="1.0" encoding="ASCII"?>
2010 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2011           targetNamespace="http://foo.com"
2012           xmlns:foo="http://foo.com"
2013           name="MyValueServices" >
2014
2015     <service name="MyValueService" promote="MyValueServiceComponent">
2016       <interface.java interface="services.myvalue.MyValueService" />
2017       <binding.ws port="http://www.myvalue.org/MyValueService#
2018                 wsdl.endpoint(MyValueService/MyValueServiceSOAP)" />
2019     </service>
2020
2021 </composite>
```

2022

2023 The following snippet shows the content of the MyValueComponents.composite file.

2024

```
2025 <?xml version="1.0" encoding="ASCII"?>
2026 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2027           targetNamespace="http://foo.com"
2028           xmlns:foo="http://foo.com"
2029           name="MyValueComponents" >
2030
2031     <component name="MyValueServiceComponent">
2032       <implementation.java
2033       class="services.myvalue.MyValueServiceImpl" />
```

```

2034         <property name="currency">EURO</property>
2035     </component>
2036
2037     <component name="StockQuoteMediatorComponent">
2038         <implementation.java class="services.myvalue.SQMediatorImpl"/>
2039         <property name="currency">EURO</property>
2040     </component>
2041
2042 <composite>
2043

```

2044 The following snippet shows the content of the MyValueReferences.composite file.

```

2045
2046 <?xml version="1.0" encoding="ASCII"?>
2047 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2048     targetNamespace="http://foo.com"
2049     xmlns:foo="http://foo.com"
2050     name="MyValueReferences" >
2051
2052     <reference name="CustomerService"
2053         promote="MyValueServiceComponent/CustomerService">
2054         <interface.java interface="services.customer.CustomerService"/>
2055         <binding.sca/>
2056     </reference>
2057
2058     <reference name="StockQuoteService"
2059 promote="StockQuoteMediatorComponent">
2060         <interface.java
2061 interface="services.stockquote.StockQuoteService"/>
2062         <binding.ws port="http://www.stockquote.org/StockQuoteService#
2063     wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
2064     </reference>
2065
2066 </composite>

```

2067 The following snippet shows the content of the MyValueWires.composite file.

```

2068
2069 <?xml version="1.0" encoding="ASCII"?>
2070 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2071     targetNamespace="http://foo.com"
2072     xmlns:foo="http://foo.com"
2073     name="MyValueWires" >
2074
2075     <wire source="MyValueServiceComponent/stockQuoteService"
2076         target="StockQuoteMediatorComponent"/>

```

2077
2078

`</composite>`

2079
2080
2081

6.7 Composites which Include Component Implementations of Multiple Types

2082
2083
2084
2085

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

2086

7 ConstrainingType

2087 SCA allows a component, and its associated implementation, to be constrained by a
2088 **constrainingType**. The constrainingType element provides assistance in developing top-down
2089 usecases in SCA, where an architect or assembler can define the structure of a composite,
2090 including the required form of component implementations, before any of the implementations are
2091 developed.

2092 A constrainingType is expressed as an element which has services, reference and properties as
2093 child elements and which can have intents applied to it. The constrainingType is independent of
2094 any implementation. Since it is independent of an implementation it cannot contain any
2095 implementation-specific configuration information or defaults. Specifically, it cannot contain
2096 bindings, policySets, property values or default wiring information. The constrainingType is
2097 applied to a component through a constrainingType attribute on the component.

2098 A constrainingType provides the "shape" for a component and its implementation. Any component
2099 configuration that points to a constrainingType is constrained by this shape. The constrainingType
2100 specifies the services, references and properties that must be implemented. This provides the
2101 ability for the implementer to program to a specific set of services, references and properties as
2102 defined by the constrainingType. Components are therefore configured instances of
2103 implementations and are constrained by an associated constrainingType.

2104 If the configuration of the component or its implementation do not conform to the
2105 constrainingType, it is an error.

2106 A constrainingType is represented by a **constrainingType** element. The following snippet shows
2107 the pseudo-schema for the composite element.

2108

```
2109 <?xml version="1.0" encoding="ASCII"?>
```

```
2110 <!-- ConstrainingType schema snippet -->
```

```
2111 <constrainingType xmlns="http://docs.oasis-  
2112 open.org/ns/opencsa/sca/200712"
```

```
2113     targetNamespace="xs:anyURI"?
```

```
2114     name="xs:NCName" requires="list of xs:QName"?>
```

2115

2116

```
2117     <service name="xs:NCName" requires="list of xs:QName"?>*
```

```
2118         <interface ... />?
```

```
2119     </service>
```

2120

```
2121     <reference name="xs:NCName"
```

```
2122         multiplicity="0..1 or 1..1 or 0..n or 1..n"?
```

```
2123         requires="list of xs:QName"?>*
```

```
2124         <interface ... />?
```

```
2125     </reference>
```

2126

```
2127     <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
```

```
2128         many="xs:boolean"? mustSupply="xs:boolean"?>*
```

```
2129         default-property-value?
```

```
2130     </property>
```

2131
2132 `</constrainingType>`
2133

2134 The constrainingType element has the following *attributes*:

- 2135 • **name (required)** – the name of the constrainingType. The form of a constrainingType
2136 name is an XML QName, in the namespace identified by the targetNamespace attribute.
- 2137 • **targetNamespace (optional)** – an identifier for a target namespace into which the
2138 constrainingType is declared
- 2139 • **requires (optional)** – a list of policy intents. See [the Policy Framework specification \[10\]](#)
2140 for a description of this attribute.

2141 ConstrainingType contains *zero or more properties, services, references*.

2142
2143 When an implementation is constrained by a constrainingType it must define all the services,
2144 references and properties specified in the corresponding constrainingType. The constraining type's
2145 references and services will have interfaces specified and may have intents specified. An
2146 implementation may contain additional services, additional optional references and additional
2147 optional properties, but cannot contain additional non-optional references or additional non-
2148 optional properties (a non-optional property is one with no default value applied).

2149 When a component is constrained by a constrainingType (via the "constrainingType" attribute),
2150 the entire componentType associated with the component and its implementation is not visible to
2151 the containing composite. The containing composite can only see a projection of the
2152 componentType associated with the component and implementation as scoped by the
2153 constrainingType of the component. For example, an additional service provided by the
2154 implementation which is not in the constrainingType associated with the component cannot be
2155 promoted by the containing composite. This requirement ensures that the constrainingType
2156 contract cannot be violated by the composite.

2157 The constrainingType can include required intents on any element. Those intents are applied to
2158 any component that uses that constrainingType. In other words, if requires="reliability" exists on
2159 a constrainingType, or its child service or reference elements, then a constrained component or its
2160 implementation must include requires="reliability" on the component or implementation or on its
2161 corresponding service or reference. Note that the component or implementation may use a
2162 qualified form of an intent specified in unqualified form in the constrainingType, but if the
2163 constrainingType uses the qualified form, then the component or implementation must also use
2164 the qualified form, otherwise there is an error.

2165 A constrainingType can be applied to an implementation. In this case, the implementation's
2166 componentType has a constrainingType attribute set to the QName of the constrainingType.

2167

2168 7.1 Example constrainingType

2169

2170 The following snippet shows the contents of the component called "MyValueServiceComponent"
2171 which is constrained by the constrainingType myns:CT. The componentType associated with the
2172 implementation is also shown.

2173

```
2174 <component name="MyValueServiceComponent" constrainingType="myns:CT">  
2175   <implementation.java class="services.myvalue.MyValueServiceImpl"/>  
2176   <property name="currency">EURO</property>  
2177   <reference name="customerService" target="CustomerService">  
2178     <binding.ws ...>
```

```
2179     <reference name="StockQuoteService"
2180         target="StockQuoteMediatorComponent" />
2181 </component>
2182
2183 <constrainingType name="CT"
2184     targetNamespace="http://mysns.com">
2185     <service name="MyValueService">
2186         <interface.java interface="services.myvalue.MyValueService" />
2187     </service>
2188     <reference name="customerService">
2189         <interface.java interface="services.customer.CustomerService" />
2190     </reference>
2191     <reference name="stockQuoteService">
2192         <interface.java interface="services.stockquote.StockQuoteService" />
2193     </reference>
2194     <property name="currency" type="xsd:string" />
2195 </constrainingType>
```

2196 The component `MyValueServiceComponent` is constrained by the constrainingType `CT` which
2197 means that it must provide:

- 2198 • service ***MyValueService*** with the interface `services.myvalue.MyValueService`
- 2199 • reference ***customerService*** with the interface `services.stockquote.StockQuoteService`
- 2200 • reference ***stockQuoteService*** with the interface `services.stockquote.StockQuoteService`
- 2201 • property ***currency*** of type `xsd:string`.

2202

8 Interface

2203

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

2208

2209

SCA currently supports the following interface type systems:

2210

- Java interfaces

2211

- WSDL 1.1 portTypes

2212

- WSDL 2.0 interfaces

2213

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

2214

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

2215

2216

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

2217

2218

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

2219

2220

The interface.java element has the following attributes:

2221

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

2222

2223

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

2224

2225

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

2226

2227

Here, the Java interface is defined in the Java class file

2228

./services/stockquote/StockQuoteService.class, where the root directory is defined by the

2229

contribution in which the interface exists.

2230

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

2231

2232

2233

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

2234

2235

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

2236

2237

2238

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

2239

2240

The interface.wsdl element has the following attributes:

2241

- **interface** – URI of the portType/interface with the following format

2242

- `<WSDL-namespace-URI>#wsdl.interface(<portTypeOrInterface-name>)`

2243

2244 The following snippet shows a sample for the WSDL portType/interface element.

```
2245  
2246 <interface.wsdl interface="http://www.stockquote.org/StockQuoteService#  
2247                               wsdl.interface(StockQuo  
2248                               te)"/>  
2249
```

2250 For WSDL 1.1, the interface attribute points to a portType in the WSDL. For WSDL 2.0, the
2251 interface attribute points to an interface in the WSDL. For the WSDL 1.1 portType and WSDL 2.0
2252 interface type systems, arguments and return of the service operations are described using XML
2253 schema.

2254

2255

2256 8.1 Local and Remotable Interfaces

2257 A remotable service is one which may be called by a client which is running in an operating system
2258 process different from that of the service itself (this also applies to clients running on different
2259 machines from the service). Whether a service of a component implementation is remotable is
2260 defined by the interface of the service. In the case of Java this is defined by adding the
2261 **@Remotable** annotation to the Java interface (see [Client and Implementation Model Specification
2262 for Java](#)). WSDL defined interfaces are always remotable.

2263

2264 The style of remotable interfaces is typically **coarse grained** and intended for **loosely coupled**
2265 interactions. Remotable service Interfaces MUST NOT make use of **method or operation**
2266 **overloading**.

2267

2268 Independent of whether the remotable service is called remotely from outside the process where
2269 the service runs or from another component running in the same process, the data exchange
2270 semantics are **by-value**.

2271 Implementations of remotable services may modify input messages (parameters) during or after
2272 an invocation and may modify return messages (results) after the invocation. If a remotable
2273 service is called locally or remotely, the SCA container is responsible for making sure that no
2274 modification of input messages or post-invocation modifications to return messages are seen by
2275 the caller.

2276 Here is a snippet which shows an example of a remotable java interface:

2277

```
2278 package services.hello;  
2279  
2280 @Remotable  
2281 public interface HelloService {  
2282  
2283     String hello(String message);  
2284 }
```

2285

2286 It is possible for the implementation of a remotable service to indicate that it can be called using
2287 by-reference data exchange semantics when it is called from a component in the same process.
2288 This can be used to improve performance for service invocations between components that run in
2289 the same process. This can be done using the @AllowsPassByReference annotation (see the [Java
2290 Client and Implementation Specification](#)).

2291

2292 A service typed by a local interface can only be called by clients that are running in the same
2293 process as the component that implements the local service. Local services cannot be published
2294 via remotable services of a containing composite. In the case of Java a local service is defined by a
2295 Java interface definition without a *@Remotable* annotation.

2296

2297 The style of local interfaces is typically *fine grained* and intended for *tightly coupled*
2298 interactions. Local service interfaces can make use of *method or operation overloading*.

2299 The data exchange semantic for calls to services typed by local interfaces is *by-reference*.

2300

2301 8.2 Bidirectional Interfaces

2302 The relationship of a business service to another business service is often peer-to-peer, requiring
2303 a two-way dependency at the service level. In other words, a business service represents both a
2304 consumer of a service provided by a partner business service and a provider of a service to the
2305 partner business service. This is especially the case when the interactions are based on
2306 asynchronous messaging rather than on remote procedure calls. The notion of *bidirectional*
2307 *interfaces* is used in SCA to directly model peer-to-peer bidirectional business service
2308 relationships.

2309 An interface element for a particular interface type system must allow the specification of an
2310 optional callback interface. If a callback interface is specified SCA refers to the interface as a whole
2311 as a bidirectional interface.

2312 The following snippet shows the interface element defined using Java interfaces with an optional
2313 callbackInterface attribute.

2314

```
2315 <interface.java          interface="services.invoicing.ComputePrice"  
2316                       callbackInterface="services.invoicing.InvoiceCallback"/>
```

2317

2318 If a service is defined using a bidirectional interface element then its implementation implements
2319 the interface, and its implementation uses the callback interface to converse with the client that
2320 called the service interface.

2321

2322 If a reference is defined using a bidirectional interface element, the client component
2323 implementation using the reference calls the referenced service using the interface. The client
2324 must provide an implementation of the callback interface.

2325 Callbacks may be used for both remotable and local services. Either both interfaces of a
2326 bidirectional service MUST be remotable, or both MUST be local. A bidirectional service MUST NOT
2327 mix local and remote services.

2328

2329 8.3 Conversational Interfaces

2330

2331 Services sometimes cannot easily be defined so that each operation stands alone and is
2332 completely independent of the other operations of the same service. Instead, there is a sequence
2333 of operations that must be called in order to achieve some higher level goal. SCA calls this
2334 sequence of operations a *conversation*. If the service uses a bidirectional interface, the
2335 conversation may include both operations and callbacks.

2336

2337 Such conversational services are typically managed by using conversation identifiers that are
2338 either (1) part of the application data (message parts or operation parameters) or 2)
2339 communicated separately from application data (possibly in headers). SCA introduces the concept
2340 of *conversational interfaces* for describing the interface contract for conversational services of the
2341 second form above. With this form, it is possible for the runtime to automatically manage the
2342 conversation, with the help of an appropriate binding specified at deployment. SCA does not
2343 standardize any aspect of conversational services that are maintained using application data.
2344 Such services are neither helped nor hindered by SCA's conversational service support.

2345

2346 Conversational services typically involve state data that relates to the conversation that is taking
2347 place. The creation and management of the state data for a conversation has a significant impact
2348 on the development of both clients and implementations of conversational services.

2349

2350 Traditionally, application developers who have needed to write conversational services have been
2351 required to write a lot of plumbing code. They need to:

2352

- 2353 - choose or define a protocol to communicate conversational (correlation) information
2354 between the client & provider
- 2355 - route conversational messages in the provider to a machine that can handle that
2356 conversation, while handling concurrent data access issues
- 2357 - write code in the client to use/encode the conversational information
- 2358 - maintain state that is specific to the conversation, sometimes persistently and
2359 transactionally, both in the implementation and the client.

2360

2361 SCA makes it possible to divide the effort associated with conversational services between a
2362 number of roles:

- 2363 - Application Developer: Declares that a service interface is conversational (leaving the
2364 details of the protocol up to the binding). Uses lifecycle semantics, APIs or other
2365 programmatic mechanisms (as defined by the implementation-type being used) to
2366 manage conversational state.
- 2367 - Application Assembler: chooses a binding that can support conversations
- 2368 - Binding Provider: implements a protocol that can pass conversational information with
2369 each operation request/response.
- 2370 - Implementation-Type Provider: defines APIs and/or other programmatic mechanisms for
2371 application developers to access conversational information. Optionally implements
2372 instance lifecycle semantics that automatically manage implementation state based on
2373 the binding's conversational information.

2374

2375 This specification requires interfaces to be marked as conversational by means of a policy intent
2376 with the name "**conversational**". The form of the marking of this intent depends on the
2377 interface type. Note that it is also possible for a service or a reference to set the conversational
2378 intent when using an interface which is not marked with the conversational intent. This can be
2379 useful when reusing an existing interface definition that does not contain SCA information.

2380 The meaning of the conversational intent is that both the client and the provider of the interface
2381 may assume that messages (in either direction) will be handled as part of an ongoing conversation
2382 without depending on identifying information in the body of the message (i.e. in parameters of the
2383 operations). In effect, the conversation interface specifies a high-level abstract protocol that must
2384 be satisfied by any actual binding/policy combination used by the service.

2385 Examples of binding/policy combinations that support conversational interfaces are:

- 2386 - Web service binding with a WS-RM policy
- 2387 - Web service binding with a WS-Addressing policy
- 2388 - Web service binding with a WS-Context policy
- 2389 - JMS binding with a conversation policy that uses the JMS correlationID header

2390

2391 Conversations occur between one client and one target service. Consequently, requests originating
2392 from one client to multiple target conversational services will result in multiple conversations. For
2393 example, if a client A calls services B and C, both of which implement conversational interfaces,
2394 two conversations result, one between A and B and another between A and C. Likewise, requests
2395 flowing through multiple implementation instances will result in multiple conversations. For
2396 example, a request flowing from A to B and then from B to C will involve two conversations (A and
2397 B, B and C). In the previous example, if a request was then made from C to A, a third
2398 conversation would result (and the implementation instance for A would be different from the one
2399 making the original request).

2400 Invocation of any operation of a conversational interface MAY start a conversation. The decision on
2401 whether an operation would start a conversation depends on the component's implementation and
2402 its implementation type. Implementation types MAY support components with conversational
2403 services. If an implementation type does provide this support, it must provide a mechanism for
2404 determining when a new conversation should be used for an operation (for example, in Java, the
2405 conversation is new on the first use of an injected reference; in BPEL, the conversation is new
2406 when the client's partnerLink comes into scope).

2407

2408 One or more operations in a conversational interface may be annotated with an *endsConversation*
2409 annotation (the mechanism for annotating the interface depends on the interface type). Where an
2410 interface is **bidirectional**, operations may also be annotated in this way on operations of a
2411 callback interface. When a conversation ending operation is called, it indicates to both the client
2412 and the service provider that the conversation is complete. Any subsequent attempts to call an
2413 operation or a callback operation associated with the same conversation will generate a
2414 `sca:ConversationViolation` fault.

2415 A `sca:ConversationViolation` fault is thrown when one of the following errors occur:

- 2416 - A message is received for a particular conversation, after the conversation has ended
- 2417 - The conversation identification is invalid (not unique, out of range, etc.)
- 2418 - The conversation identification is not present in the input message of the operation that
2419 ends the conversation
- 2420 - The client or the service attempts to send a message in a conversation, after the
2421 conversation has ended

2422 This fault is named within the SCA namespace standard prefix "sca", which corresponds to URI
2423 <http://docs.oasis-open.org/ns/opencsa/sca/200712>.

2424 The lifecycle of resources and the association between unique identifiers and conversations are
2425 determined by the service's implementation type and may not be directly affected by the
2426 "endConversation" annotation. For example, a **WS-BPEL** process **may** outlive most of the
2427 conversations that it is involved in.

2428 Although conversational interfaces do not require that any identifying information be passed as
2429 part of the body of messages, there is conceptually an identity associated with the conversation.
2430 Individual implementations types MAY provide an API to access the ID associated with the
2431 conversation, although no assumptions may be made about the structure of that identifier.
2432 Implementation types MAY also provide a means to set the conversation ID by either the client or
2433 the service provider, although the operation may only be supported by some binding/policy
2434 combinations.

2435

2436 Implementation-type specifications are encouraged to define and provide conversational instance
2437 lifecycle management for components that implement conversational interfaces. However,
2438 implementations may also manage the conversational state manually.

2439

2440 8.4 SCA-Specific Aspects for WSDL Interfaces

2441 There are a number of aspects that SCA applies to interfaces in general, such as marking them
2442 **conversational**. These aspects apply to the interfaces themselves, rather than their use in a
2443 specific place within SCA. There is thus a need to provide appropriate ways of marking the
2444 interface definitions themselves, which go beyond the basic facilities provided by the interface
2445 definition language.

2446 For WSDL interfaces, there is an extension mechanism that permits additional information to be
2447 included within the WSDL document. SCA takes advantage of this extension mechanism. In order
2448 to use the SCA extension mechanism, the SCA namespace ([http://docs.oasis-](http://docs.oasis-open.org/ns/opencsa/sca/200712)
2449 [open.org/ns/opencsa/sca/200712](http://docs.oasis-open.org/ns/opencsa/sca/200712)) must be declared within the WSDL document.

2450 First, SCA defines a global attribute in the SCA namespace which provides a mechanism to attach
2451 policy intents - **@requires**. The definition of this attribute is as follows:

```
2452 <attribute name="requires" type="sca:listOfQNames"/>
```

2453

```
2454 <simpleType name="listOfQNames">
```

```
2455 <list itemType="QName"/>
```

```
2456 </simpleType>
```

2457 The @requires attribute can be applied to WSDL Port Type elements (WSDL 1.1) and to WSDL
2458 Interface elements (WSDL 2.0). The attribute contains one or more intent names, as defined by
2459 the [Policy Framework specification \[10\]](#). Any service or reference that uses an interface with
2460 required intents implicitly adds those intents to its own @requires list.

2461 To specify that a WSDL interface is conversational, the following attribute setting is used on either
2462 the WSDL Port Type or WSDL Interface:

```
2463 requires="conversational"
```

2464 SCA defines an **endsConversation** attribute that is used to mark specific operations within a
2465 WSDL interface declaration as ending a conversation. This only has meaning for WSDL interfaces
2466 which are also marked conversational. The endsConversation attribute is a global attribute in the
2467 SCA namespace, with the following definition:

```
2468 <attribute name="endsConversation" type="boolean" default="false"/>
```

2469

2470 The following snippet is an example of a WSDL Port Type annotated with the **requires** attribute on
2471 the portType and the **endsConversation** attribute on one of the operations:

2472

```
...
```

```
2473 <portType name="LoanService" sca:requires="conversational">
```

```
2474 <operation name="apply">
```

```
2475 <input message="tns:ApplicationInput"/>
```

```
2476 <output message="tns:ApplicationOutput"/>
```

```
2477 </operation>
```

```
2478 <operation name="cancel" sca:endsConversation="true">
```

```
2479 </operation>
```

2480

```
...
```

```
2481 </portType>
```

2482

```
...
```


2483

9 Binding

2484

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

2485

Services use bindings to describe the access mechanism that clients (which can be a client from another SCA composite) have to use to call the service.

2486

2487

2488

2489

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

2490

2491

2492

2493

2494

2495

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

2496

2497

2498

2499

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

2500

```
<!-- Bindings schema snippet -->
```

2501

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

2502

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

2503

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

2504

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

2505

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

2506

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

2507

2508

```
...
```

2509

2510

```
<service name="xs:NCName" promote="xs:anyURI"
```

2511

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

2512

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

2513

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

2514

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

2515

```
xs:QName"?/>*
```

2516

```
<callback?>
```

2517

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

2518

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

2519

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

2520

```
</callback>
```

2521

```
</service>
```

2522

2523

```
...
```

2524

2525

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

```

2526     promote="list of xs:anyURI"? wiredByImpl="xs:boolean"?
2527     multiplicity="0..1 or 1..1 or 0..n or 1..n"?
2528     requires="list of xs:QName"? policySets="list of xs:QName"?>+*
2529     <interface ... />?
2530     <binding uri="xs:anyURI"? name="xs:NCName"?
2531         requires="list of xs:QName"? policySets="list of
2532 xs:QName"?/>+*
2533     <callback>?
2534         <binding uri="xs:anyURI"? name="xs:NCName"?
2535             requires="list of xs:QName"?
2536             policySets="list of xs:QName"?/>+
2537     </callback>
2538 </reference>
2539
2540     ...
2541
2542 </composite>
2543

```

2544 The element name of the binding element is architected; it is in itself a qualified name. The first
2545 qualifier is always named "binding", and the second qualifier names the respective binding-type
2546 (e.g. binding.composite, binding.ws, binding.ejb, binding.eis).

2547

2548 A binding element has the following attributes:

- 2549
- 2550 • **uri (optional)** - has the following semantic.
 - 2551 ○ For a binding of a **reference** the URI attribute defines the target URI of the
 - 2552 reference (either the component/service for a wire to an endpoint within the SCA
 - 2553 domain or the accessible address of some endpoint outside the SCA domain). It is
 - 2554 optional for references defined in composites used as component implementations,
 - 2555 but required for references defined in composites contributed to SCA domains. The
 - 2556 URI attribute of a reference of a composite can be reconfigured by a component in
 - 2557 a containing composite using the composite as an implementation. Some binding
 - 2558 types may require that the address of the target service uses more than a simple
 - 2559 URI (such as a WS-Addressing endpoint reference). In those cases, the binding
 - 2560 type will define the additional attributes or sub-elements that are necessary to
 - 2561 identify the service.
 - 2562 ○ For a binding of a **service** the URI attribute defines the URI relative to the
 - 2563 component which contributes the service to the SCA domain. The default value for
 - 2564 the URI is the the value of the name attribute of the binding.
 - 2564 • **name (optional)** – a name for the binding instance (an NCName). The name attribute
 - 2565 allows distinction between multiple binding elements on a single service or reference. The
 - 2566 default value of the name attribute is the service or reference name. When a service or
 - 2567 reference has multiple bindings, only one can have the default value; all others must have
 - 2568 a value specified that is unique within the service or reference. The name also permits the
 - 2569 binding instance to be referenced from elsewhere – particularly useful for some types of
 - 2570 binding, which can be declared in a definitions document as a template and referenced
 - 2571 from other binding instances, simplifying the definition of more complex binding instances
 - 2572 (see [the JMS Binding specification \[11\]](#) for examples of this referencing).
 - 2573 • **requires (optional)** - a list of policy intents. See the [Policy Framework specification \[10\]](#)
 - 2574 for a description of this attribute.

2575 • **policySets (optional)** – a list of policy sets. See the [Policy Framework specification \[10\]](#)
2576 for a description of this attribute.

2577 When multiple bindings exist for an service, it means that the service is available by any of the
2578 specified bindings. The technique that the SCA runtime uses to choose among available bindings
2579 is left to the implementation and it may include additional (nonstandard) configuration. Whatever
2580 technique is used SHOULD be documented.

2581 Services and References can always have their bindings overridden at the SCA domain level,
2582 unless restricted by Intents applied to them.

2583 The following sections describe the SCA and Web service binding type in detail.

2584

2585 9.1 Messages containing Data not defined in the Service Interface

2586

2587 It is possible for a message to include information that is not defined in the interface used to
2588 define the service, for instance information may be contained in SOAP headers or as MIME
2589 attachments.

2590 Implementation types MAY make this information available to component implementations in their
2591 execution context. These implementation types must indicate how this information is accessed
2592 and in what form they are presented.

2593

2594 9.2 Form of the URI of a Deployed Binding

2595

2596 9.2.1 Constructing Hierarchical URIs

2597 Bindings that use hierarchical URI schemes construct the effective URI with a combination of the
2598 following pieces:

2599 Base System URI for a scheme / Component URI / Service Binding URI

2600

2601 Each of these components deserves addition definition:

2602 **Base Domain URI for a scheme.** An SCA domain should define a base URI for each hierarchical
2603 URI scheme on which it intends to provide services.

2604 For example: the HTTP and HTTPS schemes would each have their own base URI defined for the
2605 domain. An example of a scheme that is not hierarchical, and therefore will have no base URI is
2606 the "jms:" scheme.

2607 **Component URI.** The component URI above is for a component that is deployed in the SCA
2608 Domain. The URI of a component defaults to the name of the component, which is used as a
2609 relative URI. The component may have a specified URI value. The specified URI value may be an
2610 absolute URI in which case it becomes the Base URI for all the services belonging to the
2611 component. If the specified URI value is a relative URI, it is used as the Component URI value
2612 above.

2613 **Service Binding URI.** The Service Binding URI is the relative URI specified in the "uri" attribute
2614 of a binding element of the service. The default value of the attribute is value of the binding's
2615 name attribute treated as a relative URI. If multiple bindings for a single service use the same
2616 scheme (e.g. HTTP), then only one of the bindings may depend on the default value for the uri
2617 attribute, i.e. only one may use the default binding name. The service binding URI may also be
2618 absolute, in which case the absolute URI fully specifies the full URI of the service. Some
2619 deployment environments may not support the use of absolute URIs in service bindings.

2620 Services deployed into the Domain (as opposed to services of components) have a URI that does
2621 not include a component name, i.e.:

2622 Base Domain URI for a scheme / Service Binding URI

2623 The name of the containing composite does not contribute to the URI of any service.

2624 For example, a service where the Base URI is "http://acme.com", the component is named
2625 "stocksComponent" and the service binding name is "getQuote", the URI would look like this:

2626 http://acme.com/stocksComponent/getQuote

2627 Allowing a binding's relative URI to be specified that differs from the name of the service allows
2628 the URI hierarchy of services to be designed independently of the organization of the domain.

2629 It is good practice to design the URI hierarchy to be independent of the domain organization, but
2630 there may be times when domains are initially created using the default URI hierarchy. When this
2631 is the case, the organization of the domain can be changed, while maintaining the form of the URI
2632 hierarchy, by giving appropriate values to the *uri* attribute of select elements. Here is an example
2633 of a change that can be made to the organization while maintaining the existing URIs:

2634 To move a subset of the services out of one component (say "foo") to a new component (say
2635 "bar"), the new component should have bindings for the moved services specify a URI
2636 "../foo/MovedService"..

2637 The URI attribute may also be used in order to create shorter URIs for some endpoints, where the
2638 component name may not be present in the URI at all. For example, if a binding has a *uri*
2639 attribute of "../myService" the component name will not be present in the URI.

2640 9.2.2 Non-hierarchical URIs

2641 Bindings that use non-hierarchical URI schemes (such as jms: or mailto:) may optionally make
2642 use of the "uri" attribute, which is the complete representation of the URI for that service
2643 binding. Where the binding does not use the "uri" attribute, the binding must offer a different
2644 mechanism for specifying the service address.

2645 9.2.3 Determining the URI scheme of a deployed binding

2646 One of the things that needs to be determined when building the effective URI of a deployed
2647 binding (i.e. endpoint) is the URI scheme. The process of determining the endpoint URI scheme is
2648 binding type specific.

2649 If the binding type supports a single protocol then there is only one URI scheme associated with it.
2650 In this case, that URI scheme is used.

2651 If the binding type supports multiple protocols, the binding type implementation determines the
2652 URI scheme by introspecting the binding configuration, which may include the policy sets
2653 associated with the binding.

2654 A good example of a binding type that supports multiple protocols is binding.ws, which can be
2655 configured by referencing either an "abstract" WSDL element (i.e. portType or interface) or a
2656 "concrete" WSDL element (i.e. binding, port or endpoint). When the binding references a PortType
2657 or Interface, the protocol and therefore the URI scheme is derived from the intents/policy sets
2658 attached to the binding. When the binding references a "concrete" WSDL element, there are two
2659 cases:

- 2660 1) The referenced WSDL binding element uniquely identifies a URI scheme. This is the most
2661 common case. In this case, the URI scheme is given by the protocol/transport specified in the
2662 WSDL binding element.
- 2663 2) The referenced WSDL binding element doesn't uniquely identify a URI scheme. For example,
2664 when HTTP is specified in the @transport attribute of the SOAP binding element, both "http"
2665 and "https" could be used as valid URI schemes. In this case, the URI scheme is determined
2666 by looking at the policy sets attached to the binding.

2667 It's worth noting that an intent supported by a binding type may completely change the behavior
2668 of the binding. For example, when the intent "confidentiality/transport" is required by an HTTP
2669 binding, SSL is turned on. This basically changes the URI scheme of the binding from "http" to
2670 "https".

2671

2672 9.3 SCA Binding

2673 The SCA binding element is defined by the following schema.

2674

```
2675 <binding.sca />
```

2676

2677 The SCA binding can be used for service interactions between references and services contained
2678 within the SCA domain. The way in which this binding type is implemented is not defined by the
2679 SCA specification and it can be implemented in different ways by different SCA runtimes. The only
2680 requirement is that the required qualities of service must be implemented for the SCA binding
2681 type. The SCA binding type is **not** intended to be an interoperable binding type. For
2682 interoperability, an interoperable binding type such as the Web service binding should be used.

2683 A service definition with no binding element specified uses the SCA binding.

2684 <binding.sca/> would only have to be specified in override cases, or when you specify a
2685 set of bindings on a service definition and the SCA binding should be one of them.

2686 If a reference does not have a binding, then the binding used can be any of the bindings
2687 specified by the service provider, as long as the intents required by the reference and
2688 the service are all respected.

2689 If the interface of the service or reference is local, then the local variant of the SCA
2690 binding will be used. If the interface of the service or reference is remotable, then either
2691 the local or remote variant of the SCA binding will be used depending on whether source
2692 and target are co-located or not.

2693 If a reference specifies an URI via its uri attribute, then this provides the default wire to a service
2694 provided by another domain level component. The value of the URI has to be as follows:

- 2695 • <domain-component-name>/<service-name>

2696

2697 9.3.1 Example SCA Binding

2698 The following snippet shows the MyValueComposite.composite file for the MyValueComposite
2699 containing the service element for the MyValueService and a reference element for the
2700 StockQuoteService. Both the service and the reference use an SCA binding. The target for the
2701 reference is left undefined in this binding and would have to be supplied by the composite in which
2702 this composite is used.

2703

```
2704 <?xml version="1.0" encoding="ASCII"?>
```

```
2705 <!-- Binding SCA example -->
```

```
2706 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
```

```
2707 targetNamespace="http://foo.com"
```

```
2708 name="MyValueComposite" >
```

2709

```
2710 <service name="MyValueService" promote="MyValueComponent">
```

```
2711 <interface.java interface="services.myvalue.MyValueService"/>
```

```
2712         <binding.sca/>
2713     ...
2714 </service>
2715
2716 ...
2717
2718     <reference name="StockQuoteService"
2719 promote="MyValueComponent/StockQuoteReference">
2720         <interface.java
2721 interface="services.stockquote.StockQuoteService"/>
2722         <binding.sca/>
2723     </reference>
2724
2725 </composite>
2726
```

2727 **9.4 Web Service Binding**

2728 SCA defines a Web services binding. This is described in [a separate specification document \[9\]](#).

2729

2730 **9.5 JMS Binding**

2731 SCA defines a JMS binding. This is described in [a separate specification document \[11\]](#).

2732

10 SCA Definitions

2733 There are a variety of SCA artifacts which are generally useful and which are not specific to a
2734 particular composite or a particular component. These shared artifacts include intents, policy sets,
2735 bindings, binding type definitions and implementation type definitions.

2736 All of these artifacts within an SCA Domain are defined in a global, SCA Domain-wide file named
2737 definitions.xml. The definitions.xml file contains a definitions element that conforms to the
2738 following pseudo-schema snippet:

```
2739 <?xml version="1.0" encoding="ASCII"?>
2740 <!-- Composite schema snippet -->
2741 <definitions xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2742             targetNamespace="xs:anyURI">
2743
2744     <sca:intent/*>
2745
2746     <sca:policySet/*>
2747
2748     <sca:binding/*>
2749
2750     <sca:bindingType/*>
2751
2752     <sca:implementationType/*>
2753
2754 </definitions>
```

2755 The definitions element has the following attribute:

- 2756 • **targetNamespace (required)** – the namespace into which the child elements of this
2757 definitions element are placed (used for artifact resolution)

2758 The definitions element contains optional child elements – intent, policySet, binding, bindingtype
2759 and implementationType. These elements are described elsewhere in this specification or in [the](#)
2760 [SCA Policy Framework specification \[10\]](#). The use of the elements declared within a definitions
2761 element is described in the SCA Policy Framework specification [10] and in [the JMS Binding](#)
2762 [specification \[11\]](#).

2763

2764

11 Extension Model

2765

2766 The assembly model can be extended with support for new interface types, implementation types
2767 and binding types. The extension model is based on XML schema substitution groups. There are
2768 three XML Schema substitution group heads defined in the SCA namespace: **interface**,
2769 **implementation** and **binding**, for interface types, implementation types and binding types,
2770 respectively.

2771 The SCA Client and Implementation specifications and the SCA Bindings specifications (see [1])
2772 use these XML Schema substitution groups to define some basic types of interfaces,
2773 implementations and bindings, but other types can be defined as required, where support for
2774 these extra ones is available from the runtime. The interface type elements, implementation type
2775 elements, and binding type elements defined by the SCA specifications (see [1]) are all part of the
2776 SCA namespace ("http://docs.oasis-open.org/ns/opencsa/sca/200712"), as indicated in their
2777 respective schemas. New interface types, implementation types and binding types that are defined
2778 using this extensibility model, which are not part of these SCA specifications must be defined in
2779 namespaces other than the SCA namespace.

2780 The "." notation is used in naming elements defined by the SCA specifications (e.g.
2781 <implementation.java ... />, <interface.wsdl ... />, <binding.ws ... />), not as a parallel
2782 extensibility approach but as a naming convention that improves usability of the SCA assembly
2783 language.

2784

2785 **Note:** How to contribute SCA model extensions and their runtime function to an SCA runtime will
2786 be defined by a future version of the specification.

2787

11.1 Defining an Interface Type

2788 The following snippet shows the base definition for the **interface** element and **Interface** type
2789 contained in **sca-core.xsd**; see appendix for complete schema.

2790

```
2791
2792 <?xml version="1.0" encoding="UTF-8"?>
2793 <!-- (c) Copyright SCA Collaboration 2006 -->
2794 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2795         targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2796         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2797         elementFormDefault="qualified">
2798
2799     ...
2800
2801     <element name="interface" type="sca:Interface" abstract="true"/>
2802     <complexType name="Interface"/>
2803
2804     ...
2805
2806 </schema>
```


2807 In the following snippet we show how the base definition is extended to support Java interfaces.
2808 The snippet shows the definition of the *interface.java* element and the *JavaInterface* type
2809 contained in *sca-interface-java.xsd*.

2810

```
2811 <?xml version="1.0" encoding="UTF-8"?>
2812 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2813         targetNamespace="http://docs.oasis-
2814 open.org/ns/opencsa/sca/200712"
2815         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712">
2816
2817     <element name="interface.java" type="sca:JavaInterface"
2818             substitutionGroup="sca:interface"/>
2819     <complexType name="JavaInterface">
2820         <complexContent>
2821             <extension base="sca:Interface">
2822                 <attribute name="interface" type="NCName"
2823 use="required"/>
2824             </extension>
2825         </complexContent>
2826     </complexType>
2827 </schema>
```

2828 In the following snippet we show an example of how the base definition can be extended by other
2829 specifications to support a new interface not defined in the SCA specifications. The snippet shows
2830 the definition of the *my-interface-extension* element and the *my-interface-extension-type*
2831 type.

```
2832 <?xml version="1.0" encoding="UTF-8"?>
2833 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2834         targetNamespace="http://www.example.org/myextension"
2835         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2836         xmlns:tns="http://www.example.org/myextension">
2837
2838     <element name="my-interface-extension" type="tns:my-interface-
2839 extension-type"
2840             substitutionGroup="sca:interface"/>
2841     <complexType name="my-interface-extension-type">
2842         <complexContent>
2843             <extension base="sca:Interface">
2844                 ...
2845             </extension>
2846         </complexContent>
2847     </complexType>
2848 </schema>
2849
```

2850 11.2 Defining an Implementation Type

2851 The following snippet shows the base definition for the *implementation* element and
2852 *Implementation* type contained in *sca-core.xsd*; see appendix for complete schema.

```
2853  
2854 <?xml version="1.0" encoding="UTF-8"?>  
2855 <!-- (c) Copyright SCA Collaboration 2006 -->  
2856 <schema xmlns="http://www.w3.org/2001/XMLSchema"  
2857         targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
2858         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
2859         elementFormDefault="qualified">  
2860  
2861     ...  
2862  
2863     <element name="implementation" type="sca:Implementation"  
2864     abstract="true"/>  
2865     <complexType name="Implementation"/>  
2866  
2867     ...  
2868  
2869 </schema>
```

2870
2871 In the following snippet we show how the base definition is extended to support Java
2872 implementation. The snippet shows the definition of the *implementation.java* element and the
2873 *JavaImplementation* type contained in *sca-implementation-java.xsd*.

```
2874  
2875 <?xml version="1.0" encoding="UTF-8"?>  
2876 <schema xmlns="http://www.w3.org/2001/XMLSchema"  
2877         targetNamespace="http://docs.oasis-  
2878 open.org/ns/opencsa/sca/200712"  
2879         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712">  
2880  
2881     <element name="implementation.java" type="sca:JavaImplementation"  
2882             substitutionGroup="sca:implementation"/>  
2883     <complexType name="JavaImplementation">  
2884         <complexContent>  
2885             <extension base="sca:Implementation">  
2886                 <attribute name="class" type="NCName"  
2887 use="required"/>  
2888             </extension>  
2889         </complexContent>  
2890     </complexType>  
2891 </schema>
```

2892 In the following snippet we show an example of how the base definition can be extended by other
2893 specifications to support a new implementation type not defined in the SCA specifications. The

2894 snippet shows the definition of the *my-impl-extension* element and the *my-impl-extension-*
2895 *type* type.

```
2896 <?xml version="1.0" encoding="UTF-8"?>
2897 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2898         targetNamespace="http://www.example.org/myextension"
2899         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2900         xmlns:tns="http://www.example.org/myextension">
2901
2902     <element name="my-impl-extension" type="tns:my-impl-extension-type"
2903           substitutionGroup="sca:implementation"/>
2904     <complexType name="my-impl-extension-type">
2905         <complexContent>
2906             <extension base="sca:Implementation">
2907                 ...
2908             </extension>
2909         </complexContent>
2910     </complexType>
2911 </schema>
2912
```

2913 In addition to the definition for the new implementation instance element, there needs to be an
2914 associated implementationType element which provides metadata about the new implementation
2915 type. The pseudo schema for the implementationType element is shown in the following snippet:

```
2916 <implementationType type="xs:QName"
2917                   alwaysProvides="list of intent xs:QName"
2918                   mayProvide="list of intent xs:QName"/>
2919
```

2920 The implementation type has the following attributes:

- 2921 • **type (required)** – the type of the implementation to which this implementationType
2922 element applies. This is intended to be the QName of the implementation element for the
2923 implementation type, such as "sca:implementation.java"
- 2924 • **alwaysProvides (optional)** – a set of intents which the implementation type always
2925 provides. See [the Policy Framework specification \[10\]](#) for details.
- 2926 • **mayProvide (optional)** – a set of intents which the implementation type may provide.
2927 See [the Policy Framework specification \[10\]](#) for details.

2928

2929 11.3 Defining a Binding Type

2930 The following snippet shows the base definition for the *binding* element and *Binding* type
2931 contained in *sca-core.xsd*; see appendix for complete schema.

```
2932
2933 <?xml version="1.0" encoding="UTF-8"?>
2934 <!-- binding type schema snippet -->
2935 <!-- (c) Copyright SCA Collaboration 2006, 2007 -->
2936 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2937         targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712">
```

```

2938     xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2939     elementFormDefault="qualified">
2940
2941     ...
2942
2943     <element name="binding" type="sca:Binding" abstract="true"/>
2944     <complexType name="Binding">
2945         <attribute name="uri" type="anyURI" use="optional"/>
2946         <attribute name="name" type="NCName" use="optional"/>
2947         <attribute name="requires" type="sca:listOfQNames"
2948 use="optional"/>
2949         <attribute name="policySets" type="sca:listOfQNames"
2950 use="optional"/>
2951     </complexType>
2952
2953     ...
2954
2955 </schema>

```

2956 In the following snippet we show how the base definition is extended to support Web service
2957 binding. The snippet shows the definition of the *binding.ws* element and the
2958 *WebServiceBinding* type contained in *sca-binding-webservice.xsd*.

```

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

```

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

```

2979 <?xml version="1.0" encoding="UTF-8"?>
2980 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2981     targetNamespace="http://www.example.org/myextension"
2982     xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712">

```

```

2983         xmlns:tns="http://www.example.org/myextension">
2984
2985         <element name="my-binding-extension" type="tns:my-binding-extension-
2986 type"
2987         substitutionGroup="sca:binding"/>
2988         <complexType name="my-binding-extension-type">
2989             <complexContent>
2990                 <extension base="sca:Binding">
2991                     ...
2992                 </extension>
2993             </complexContent>
2994         </complexType>
2995 </schema>
2996

```

2997 In addition to the definition for the new binding instance element, there needs to be an associated
2998 bindingType element which provides metadata about the new binding type. The pseudo schema
2999 for the bindingType element is shown in the following snippet:

```

3000 <bindingType type="xs:QName"
3001         alwaysProvides="list of intent QNames"?
3002         mayProvide = "list of intent QNames"?/>
3003

```

3004 The binding type has the following attributes:

- 3005 • **type (required)** – the type of the binding to which this bindingType element applies.
3006 This is intended to be the QName of the binding element for the binding type, such as
3007 "sca:binding.ws"
- 3008 • **alwaysProvides (optional)** – a set of intents which the binding type always provides.
3009 See [the Policy Framework specification \[10\]](#) for details.
- 3010 • **mayProvide (optional)** – a set of intents which the binding type may provide. See [the](#)
3011 [Policy Framework specification \[10\]](#) for details.

3012 12 Packaging and Deployment

3013 12.1 Domains

3014 An **SCA Domain** represents a complete runtime configuration, potentially distributed over a series
3015 of interconnected runtime nodes.

3016 A single SCA domain defines the boundary of visibility for all SCA mechanisms. For example, SCA
3017 wires can only be used to connect components within a single SCA domain. Connections to
3018 services outside the domain must use binding specific mechanisms for addressing services (such
3019 as WSDL endpoint URIs). Also, SCA mechanisms such as intents and policySets can only be used
3020 in the context of a single domain. In general, external clients of a service that is developed and
3021 deployed using SCA should not be able to tell that SCA was used to implement the service – it is
3022 an implementation detail.

3023 The size and configuration of an SCA Domain is not constrained by the SCA Assembly specification
3024 and is expected to be highly variable. An SCA Domain typically represents an area of business
3025 functionality controlled by a single organization. For example, an SCA Domain may be the whole
3026 of a business, or it may be a department within a business.

3027 As an example, for the accounts department in a business, the SCA Domain might cover all
3028 finance-related functions, and it might contain a series of composites dealing with specific areas of
3029 accounting, with one for Customer accounts and another dealing with Accounts Payable.

3030 An SCA domain has the following:

- 3031 • A virtual domain-level composite whose components are deployed and running
- 3032 • A set of *installed contributions* that contain implementations, interfaces and other artifacts
3033 necessary to execute components
- 3034 • A set of logical services for manipulating the set of contributions and the virtual domain-
3035 level composite.

3036 The information associated with an SCA domain can be stored in many ways, including but not
3037 limited to a specific filesystem structure or a repository.

3038 12.2 Contributions

3039 An SCA domain may require a large number of different artifacts in order to work. These artifacts
3040 include artifacts defined by SCA and other artifacts such as object code files and interface
3041 definition files. The SCA-defined artifact types are all XML documents. The root elements of the
3042 different SCA definition documents are: `composite`, `componentType`, `constrainingType` and
3043 `definitions`. XML artifacts that are not defined by SCA but which may be needed by an SCA
3044 domain include XML Schema documents, WSDL documents, and BPEL documents. SCA
3045 constructs, like other XML-defined constructs, use XML qualified names for their identity (i.e.
3046 namespace + local name).

3047 Non-XML artifacts are also required within an SCA domain. The most obvious examples of such
3048 non-XML artifacts are Java, C++ and other programming language files necessary for component
3049 implementations. Since SCA is extensible, other XML and non-XML artifacts may also be required.

3050 SCA defines an interoperable packaging format for contributions (ZIP), as specified below. This
3051 format is not the only packaging format that an SCA runtime can use. SCA allows many different
3052 packaging formats, but requires that the ZIP format be supported. When using the ZIP format for
3053 deploying a contribution, this specification does not specify whether that format is retained after
3054 deployment. For example, a Java EE based SCA runtime may convert the ZIP package to an EAR
3055 package. SCA expects certain characteristics of any packaging:

- 3056 • It must be possible to present the artifacts of the packaging to SCA as a hierarchy of
3057 resources based off of a single root

- 3058
- A directory resource should exist at the root of the hierarchy named META-INF
- 3059
- A document should exist directly under the META-INF directory named sca-
- 3060 contribution.xml which lists the SCA Composites within the contribution that are runnable.
- 3061
- 3062 The same document also optionally lists namespaces of constructs that are defined within
- 3063 the contribution and which may be used by other contributions
- 3064 Optionally, additional elements may exist that list the namespaces of constructs that are
- 3065 needed by the contribution and which must be found elsewhere, for example in other
- 3066 contributions. These optional elements may not be physically present in the packaging,
- 3067 but may be generated based on the definitions and references that are present, or they
- 3068 may not exist at all if there are no unresolved references.
- 3069
- 3070 See the section "SCA Contribution Metadata Document" for details of the format of this
- 3071 file.

3072 To illustrate that a variety of packaging formats can be used with SCA, the following are examples

3073 of formats that might be used to package SCA artifacts and metadata (as well as other artifacts)

3074 as a contribution:

- 3075
- A filesystem directory
- 3076
- An OSGi bundle
- 3077
- A compressed directory (zip, gzip, etc)
- 3078
- A JAR file (or its variants – WAR, EAR, etc)

3079 Contributions do not contain other contributions. If the packaging format is a JAR file that

3080 contains other JAR files (or any similar nesting of other technologies), the internal files are not

3081 treated as separate SCA contributions. It is up to the implementation to determine whether the

3082 internal JAR file should be represented as a single artifact in the contribution hierarchy or whether

3083 all of the contents should be represented as separate artifacts.

3084 A goal of SCA's approach to deployment is that the contents of a contribution should not need to

3085 be modified in order to install and use the contents of the contribution in a domain.

3086

3087 12.2.1 SCA Artifact Resolution

3088 Contributions may be self-contained, in that all of the artifacts necessary to run the contents of

3089 the contribution are found within the contribution itself. However, it may also be the case that the

3090 contents of the contribution make one or many references to artifacts that are not contained

3091 within the contribution. These references may be to SCA artifacts or they may be to other

3092 artifacts such as WSDL files, XSD files or to code artifacts such as Java class files and BPEL scripts.

3093 A contribution may use some artifact-related or packaging-related means to resolve artifact

3094 references. Examples of such mechanisms include:

- 3095
- wsdlLocation and schemaLocation attributes in references to WSDL and XSD schema
- 3096 artifacts respectively
- 3097
- OSGi bundle mechanisms for resolving Java class and related resource dependencies

3098 Where present, these mechanisms must be used to resolve artifact dependencies.

3099 SCA also provides an artifact resolution mechanism. The SCA artifact resolution mechanisms are

3100 used either where no other mechanisms are available, or in cases where the mechanisms used by

3101 the various contributions in the same SCA Domain are different. An example of the latter case is

3102 where an OSGi Bundle is used for one contribution but where a second contribution used by the

3103 first one is not implemented using OSGi - eg the second contribution is a mainframe COBOL

3104 service whose interfaces are declared using WSDL which must be accessed by the first

3105 contribution.

3106 The SCA artifact resolution is likely to be most useful for SCA domains containing heterogeneous
3107 mixtures of contribution, where artifact-related or packaging-related mechanisms are unlikely to
3108 work across different kinds of contribution.

3109 SCA artifact resolution works on the principle that a contribution which needs to use artifacts
3110 defined elsewhere expresses these dependencies using *import* statements in metadata belonging
3111 to the contribution. A contribution controls which artifacts it makes available to other
3112 contributions through *export* statements in metadata attached to the contribution.

3113

3114 12.2.2 SCA Contribution Metadata Document

3115 The contribution optionally contains a document that declares runnable composites, exported
3116 definitions and imported definitions. The document is found at the path of META-INF/sca-
3117 contribution.xml relative to the root of the contribution. Frequently some SCA metadata may
3118 need to be specified by hand while other metadata is generated by tools (such as the <import>
3119 elements described below). To accommodate this, it is also possible to have an identically
3120 structured document at META-INF/sca-contribution-generated.xml. If this document exists (or is
3121 generated on an as-needed basis), it will be merged into the contents of sca-contribution.xml,
3122 with the entries in sca-contribution.xml taking priority if there are any conflicting declarations.

3123

3124 The format of the document is:

```
3125 <?xml version="1.0" encoding="ASCII"?>  
3126 <!-- sca-contribution pseudo-schema -->  
3127 <contribution xmlns=http://docs.oasis-open.org/ns/opencsa/sca/200712>  
3128  
3129     <deployable composite="xs:QName"/>*  
3130     <import namespace="xs:String" location="xs:AnyURI"?/>*  
3131     <export namespace="xs:String"/>*  
3132  
3133 </contribution>  
3134
```

3135 **deployable element:** Identifies a composite which is a composite within the contribution that is a
3136 composite intended for potential inclusion into the virtual domain-level composite. Other
3137 composites in the contribution are not intended for inclusion but only for use by other composites.
3138 New composites can be created for a contribution after it is installed, by using the [add Deployment](#)
3139 [Composite](#) capability and the add To Domain Level Composite capability.

- 3140 • **composite (required)** – The QName of a composite within the contribution.

3141

3142 **Export element:** A declaration that artifacts belonging to a particular namespace are exported
3143 and are available for use within other contributions. An export declaration in a contribution
3144 specifies a namespace, all of whose definitions are considered to be exported. By default,
3145 definitions are not exported.

3146 The SCA artifact export is useful for SCA domains containing heterogeneous mixtures of
3147 contribution packagings and technologies, where artifact-related or packaging-related mechanisms
3148 are unlikely to work across different kinds of contribution.

- 3149 • **namespace (required)** – For XML definitions, which are identified by QNames, the
3150 namespace should be the namespace URI for the exported definitions. For XML
3151 technologies that define multiple *symbol spaces* that can be used within one namespace
3152 (e.g. WSDL port types are a different symbol space from WSDL bindings), all definitions
3153 from all symbol spaces are exported.

3154

3155 Technologies that use naming schemes other than QNames must use a different export

3156 element from the same substitution group as the the SCA <export> element. The
3157 element used identifies the technology, and may use any value for the namespace that is
3158 appropriate for that technology. For example, <export.java> can be used can be used to
3159 export java definitions, in which case the namespace should be a fully qualified package
3160 name.

3161
3162 **Import element:** Import declarations specify namespaces of definitions that are needed by the
3163 definitions and implementations within the contribution, but which are not present in the
3164 contribution. It is expected that in most cases import declarations will be generated based on
3165 introspection of the contents of the contribution. In this case, the import declarations would be
3166 found in the META-INF/ sca-contribution-generated.xml document.

3167 • **namespace (required)** – For XML definitions, which are identified by QNames, the
3168 namespace should be the namespace URI for the imported definitions. For XML
3169 technologies that define multiple *symbol spaces* that can be used within one namespace
3170 (e.g. WSDL port types are a different symbol space from WSDL bindings), all definitions
3171 from all symbol spaces are imported.
3172

3173 Technologies that use naming schemes other than QNames must use a different import
3174 element from the same substitution group as the the SCA <import> element. The
3175 element used identifies the technology, and may use any value for the namespace that is
3176 appropriate for that technology. For example, <import.java> can be used can be used to
3177 import java definitions, in which case the namespace should be a fully qualified package
3178 name.

3179 • **location (optional)** – a URI to resolve the definitions for this import. SCA makes no
3180 specific requirements for the form of this URI, nor the means by which it is resolved. It
3181 may point to another contribution (through its URI) or it may point to some location
3182 entirely outside the SCA Domain.
3183

3184 It is expected that SCA runtimes may define implementation specific ways of resolving location
3185 information for artifact resolution between contributions. These mechanisms will however usually
3186 be limited to sets of contributions of one runtime technology and one hosting environment.

3187 In order to accommodate imports of artifacts between contributions of disparate runtime
3188 technologies, it is strongly suggested that SCA runtimes honor SCA contribution URIs as location
3189 specification.

3190 SCA runtimes that support contribution URIs for cross-contribution resolution of SCA artifacts
3191 should do so similarly when used as @schemaLocation and @wsdlLocation and other artifact
3192 location specifications.

3193 The order in which the import statements are specified may play a role in this mechanism. Since
3194 definitions of one namespace can be distributed across several artifacts, multiple import
3195 declarations can be made for one namespace.
3196

3197 The location value is only a default, and dependent contributions listed in the call to
3198 installContribution should override the value if there is a conflict. However, the specific
3199 mechanism for resolving conflicts between contributions that define conflicting definitions is
3200 implementation specific.

3201
3202 If the value of the location attribute is an SCA contribution URI, then the contribution packaging
3203 may become dependent on the deployment environment. In order to avoid such a dependency,
3204 dependent contributions should be specified only when deploying or updating contributions as
3205 specified in the section 'Operations for Contributions' below.

3206 12.2.3 Contribution Packaging using ZIP

3207 SCA allows many different packaging formats that SCA runtimes can support, but SCA requires
3208 that all runtimes support the ZIP packaging format for contributions. This format allows that

3209 metadata specified by the section 'SCA Contribution Metadata Document' be present. Specifically,
3210 it may contain a top-level "META-INF" directory and a "META-INF/sca-contribution.xml" file and
3211 there may also be an optional "META-INF/sca-contribution-generated.xml" file in the package. SCA
3212 defined artifacts as well as non-SCA defined artifacts such as object files, WSDL definition, Java
3213 classes may be present anywhere in the ZIP archive,

3214 A up to date definition of the ZIP file format is published by PKWARE in [an Application Note on the](#)
3215 [.ZIP file format \[12\]](#).

3216

3217 **12.3 Installed Contribution**

3218 As noted in the section above, the contents of a contribution should not need to be modified in
3219 order to install and use it within a domain. An *installed contribution* is a contribution with all of
3220 the associated information necessary in order to execute *deployable composites* within the
3221 contribution.

3222 An installed contribution is made up of the following things:

- 3223 • Contribution Packaging – the contribution that will be used as the starting point for
3224 resolving all references
- 3225 • Contribution base URI
- 3226 • Dependent contributions: a set of snapshots of other contributions that are used to resolve
3227 the import statements from the root composite and from other dependent contributions
 - 3228 ○ Dependent contributions may or may not be shared with other installed
3229 contributions.
 - 3230 ○ When the snapshot of any contribution is taken is implementation defined, ranging
3231 from the time the contribution is installed to the time of execution
- 3232 • Deployment-time composites.
3233 These are composites that are added into an installed contribution after it has been
3234 deployed. This makes it possible to provide final configuration and access to
3235 implementations within a contribution without having to modify the contribution. These
3236 are optional, as composites that already exist within the contribution may also be used for
3237 deployment.

3238

3239 Installed contributions provide a context in which to resolve qualified names (e.g. QNames in XML,
3240 fully qualified class names in Java).

3241 If multiple dependent contributions have exported definitions with conflicting qualified names, the
3242 algorithm used to determine the qualified name to use is implementation dependent.
3243 Implementations of SCA may also generate an error if there are conflicting names.

3244

3245 **12.3.1 Installed Artifact URIs**

3246 When a contribution is installed, all artifacts within the contribution are assigned URIs, which are
3247 constructed by starting with the base URI of the contribution and adding the relative URI of each
3248 artifact (recalling that SCA requires that any packaging format be able to offer up its artifacts in a
3249 single hierarchy).

3250

3251 **12.4 Operations for Contributions**

3252 SCA Domains provide the following conceptual functionality associated with contributions
3253 (meaning the function may not be represented as addressable services and also meaning that

3254 equivalent functionality may be provided in other ways). The functionality is optional meaning that
3255 some SCA runtimes may choose not to provide that functionality in any way:

3256 **12.4.1 install Contribution & update Contribution**

3257
3258 Creates or updates an installed contribution with a supplied root contribution, and installed at a
3259 supplied base URI. A supplied dependent contribution list specifies the contributions that should
3260 be used to resolve the dependencies of the root contribution and other dependent contributions.
3261 These override any dependent contributions explicitly listed via the location attribute in the import
3262 statements of the contribution.
3263

3264 SCA follows the simplifying assumption that the use of a contribution for resolving anything also
3265 means that all other exported artifacts can be used from that contribution. Because of this, the
3266 dependent contribution list is just a list of installed contribution URIs. There is no need to specify
3267 what is being used from each one.

3268 Each dependent contribution is also an installed contribution, with its own dependent
3269 contributions. By default these dependent contributions of the dependent contributions (which we
3270 will call *indirect dependent contributions*) are included as dependent contributions of the installed
3271 contribution. However, if a contribution in the dependent contribution list exports any conflicting
3272 definitions with an indirect dependent contribution, then the indirect dependent contribution is not
3273 included (i.e. the explicit list overrides the default inclusion of indirect dependent contributions).
3274 Also, if there is ever a conflict between two indirect dependent contributions, then the conflict
3275 must be resolved by an explicit entry in the dependent contribution list.

3276 Note that in many cases, the dependent contribution list can be generated. In particular, if a
3277 domain is careful to avoid creating duplicate definitions for the same qualified name, then it is
3278 easy for this list to be generated by tooling.

3279 **12.4.2 add Deployment Composite & update Deployment Composite**

3280 Adds or updates a deployment composite using a supplied composite ("composite by value" – a
3281 data structure, not an existing resource in the domain) to the contribution identified by a supplied
3282 contribution URI. The added or updated deployment composite is given a relative URI that
3283 matches the @name attribute of the composite, with a ".composite" suffix. Since all composites
3284 must run within the context of a installed contribution (any component implementations or other
3285 definitions are resolved within that contribution), this functionality makes it possible for the
3286 deployer to create a composite with final configuration and wiring decisions and add it to an
3287 installed contribution without having to modify the contents of the root contribution.

3288 Also, in some use cases, a contribution may include only implementation code (e.g. PHP scripts).
3289 It should then be possible for those to be given component names by a (possibly generated)
3290 composite that is added into the installed contribution, without having to modify the packaging.

3291 **12.4.3 remove Contribution**

3292 Removes the deployed contribution identified by a supplied contribution URI.
3293

3294 **12.5 Use of Existing (non-SCA) Mechanisms for Resolving Artifacts**

3295
3296 For certain types of artifact, there are existing and commonly used mechanisms for referencing a
3297 specific concrete location where the artifact can be resolved.

3298 Examples of these mechanisms include:

- 3299 • For WSDL files, the *@wsdlLocation* attribute is a hint that has a URI value pointing to the
3300 place holding the WSDL itself.

- 3301 • For XSDs, the *@schemaLocation* attribute is a hint which matches the namespace to a
3302 URI where the XSD is found.

3303 **Note:** In neither of these cases is the runtime obliged to use the location hint and the URI does
3304 not have to be dereferenced.

3305 SCA permits the use of these mechanisms. Where present, these mechanisms take precedence
3306 over the SCA mechanisms. However, use of these mechanisms is discouraged because tying
3307 assemblies to addresses in this way makes the assemblies less flexible and prone to errors when
3308 changes are made to the overall SCA Domain.

3309 **Note:** If one of these mechanisms is present, but there is a failure to find the resource indicated
3310 when using the mechanism (eg the URI is incorrect or invalid, say) the SCA runtime MUST raise
3311 an error and MUST NOT attempt to use SCA resolution mechanisms as an alternative.

3312

3313 12.6 Domain-Level Composite

3314 The domain-level composite is a virtual composite, in that it is not defined by a composite
3315 definition document. Rather, it is built up and modified through operations on the domain.
3316 However, in other respects it is very much like a composite, since it contains components, wires,
3317 services and references.

3318 The abstract domain-level functionality for modifying the domain-level composite is as follows,
3319 although a runtime may supply equivalent functionality in a different form:

3320 12.6.1 add To Domain-Level Composite

3321 This functionality adds the composite identified by a supplied URI to the Domain Level Composite.
3322 The supplied composite URI must refer to a composite within a installed contribution. The
3323 composite's installed contribution determines how the composite's artifacts are resolved (directly
3324 and indirectly). The supplied composite is added to the domain composite with semantics that
3325 correspond to the domain-level composite having an <include> statement that references the
3326 supplied composite. All of the composite's components become *top-level* components and the
3327 services become externally visible services (eg. they would be present in a WSDL description of
3328 the domain).

3329 12.6.2 remove From Domain-Level Composite

3330 Removes from the Domain Level composite the elements corresponding to the composite
3331 identified by a supplied composite URI. This means that the removal of the components, wires,
3332 services and references originally added to the domain level composite by the identified
3333 composite.

3334 12.6.3 get Domain-Level Composite

3335 Returns a <composite> definition that has an <include> line for each composite that had been
3336 added to the domain level composite. It is important to note that, in dereferencing the included
3337 composites, any referenced artifacts must be resolved in terms of that installed composite.

3338 12.6.4 get QName Definition

3339 In order to make sense of the domain-level composite (as returned by get Domain-Level
3340 Composite), it must be possible to get the definitions for named artifacts in the included
3341 composites. This functionality takes the supplied URI of an installed contribution (which provides
3342 the context), a supplied qualified name of a definition to look up, and a supplied symbol space (as
3343 a QName, eg wsdl:PortType). The result is a single definition, in whatever form is appropriate for
3344 that definition type.

3345 Note that this, like all the other domain-level operations, is a conceptual operation. Its capabilities
3346 should exist in some form, but not necessarily as a service operation with exactly this signature.

3347

13 Conformance

3348

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.

3349

3350

A. Pseudo Schema

3351

A.1 ComponentType

3352

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

3353

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

3354

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

3355

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

3356

3357

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

3358

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

3359

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

3360

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

3361

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

3362

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

3363

```
    <callback?>
```

3364

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

3365

```
    </callback>
```

3366

```
  </service>
```

3367

3368

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

3369

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

3370

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

3371

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

3372

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

3373

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

3374

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

3375

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

3376

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

3377

```
    <callback?>
```

3378

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

3379

```
    </callback>
```

3380

```
  </reference>
```

3381

3382

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

3383

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

3384

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

3385

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

3386

```
  </property>
```

3387

3388

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

3389

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

3390
3391 </componentType>
3392

3393 **A.2 Composite**

```
3394     <?xml version="1.0" encoding="ASCII"?>
3395     <!-- Composite schema snippet -->
3396     <composite     xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3397                   targetNamespace="xs:anyURI"
3398                   name="xs:NCName" local="xs:boolean"?
3399                   autowire="xs:boolean"? constrainingType="QName"?
3400                   requires="list of xs:QName"? policySets="list of
3401     xs:QName"?>
3402
3403         <include name="xs:QName"/>*
3404
3405         <service name="xs:NCName" promote="xs:anyURI"
3406                 requires="list of xs:QName"? policySets="list of xs:QName"?>*
3407             <interface ... />?
3408             <binding uri="xs:anyURI"? name="xs:NCName"?
3409                 requires="list of xs:QName"? policySets="list of
3410     xs:QName"?/>*
3411             <callback?
3412                 <binding uri="xs:anyURI"? name="xs:NCName"?
3413                     requires="list of xs:QName"?
3414                     policySets="list of xs:QName"?/>+
3415             </callback>
3416         </service>
3417
3418         <reference name="xs:NCName" target="list of xs:anyURI"?
3419                     promote="list of xs:anyURI" wiredByImpl="xs:boolean"?
3420                     multiplicity="0..1 or 1..1 or 0..n or 1..n"?
3421                     requires="list of xs:QName"? policySets="list of xs:QName"?>*
3422             <interface ... />?
3423             <binding uri="xs:anyURI"? name="xs:NCName"?
3424                 requires="list of xs:QName"? policySets="list of
3425     xs:QName"?/>*
3426             <callback?
3427                 <binding uri="xs:anyURI"? name="xs:NCName"?
3428                     requires="list of xs:QName"?
3429                     policySets="list of xs:QName"?/>+
3430             </callback>
3431         </reference>
3432
```



```

3433     <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
3434         many="xs:boolean"? mustSupply="xs:boolean"?>*
3435         default-property-value?
3436     </property>
3437
3438     <component name="xs:NCName" autowire="xs:boolean"?
3439         requires="list of xs:QName"? policySets="list of xs:QName"?>*
3440         <implementation ... />?
3441         <service name="xs:NCName" requires="list of xs:QName"?
3442             policySets="list of xs:QName"?>*
3443             <interface ... />?
3444             <binding uri="xs:anyURI"? name="xs:NCName"?
3445                 requires="list of xs:QName"?
3446                 policySets="list of xs:QName"?/>*
3447             <callback>?
3448                 <binding uri="xs:anyURI"? name="xs:NCName"?
3449                     requires="list of xs:QName"?
3450                     policySets="list of xs:QName"?/>+
3451             </callback>
3452         </service>
3453         <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
3454             source="xs:string"? file="xs:anyURI"?>*
3455             property-value
3456         </property>
3457         <reference name="xs:NCName" target="list of xs:anyURI"?
3458             autowire="xs:boolean"? wiredByImpl="xs:boolean"?
3459             requires="list of xs:QName"? policySets="list of xs:QName"?
3460             multiplicity="0..1 or 1..1 or 0..n or 1..n"?/>*
3461         <interface ... />?
3462         <binding uri="xs:anyURI"? name="xs:NCName"?
3463             requires="list of xs:QName"?
3464             policySets="list of xs:QName"?/>*
3465         <callback>?
3466             <binding uri="xs:anyURI"? name="xs:NCName"?
3467                 requires="list of xs:QName"?
3468                 policySets="list of xs:QName"?/>+
3469         </callback>
3470     </reference>
3471 </component>
3472
3473     <wire source="xs:anyURI" target="xs:anyURI" />*
3474
3475 </composite>

```

3476

B. XML Schemas

3477

B.1 sca.xsd

3478

3479

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

3480

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

3481

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

3482

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

3483

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

3484

3485

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

3486

3487

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

3488

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

3489

3490

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

3491

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

3492

3493

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

3494

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

3495

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

3496

3497

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

3498

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

3499

3500

```
</schema>
```

3501

3502

B.2 sca-core.xsd

3503

3504

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

3505

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

3506

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

3507

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

3508

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

3509

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

3510

3511

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

3512

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

3513

```
    <sequence>
```

3514

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

```

3515     <choice minOccurs="0" maxOccurs="unbounded">
3516         <element name="service" type="sca:ComponentService" />
3517         <element name="reference" type="sca:ComponentReference"/>
3518         <element name="property" type="sca:Property"/>
3519     </choice>
3520     <any namespace="##other" processContents="lax" minOccurs="0"
3521         maxOccurs="unbounded"/>
3522 </sequence>
3523 <attribute name="constrainingType" type="QName" use="optional"/>
3524 <anyAttribute namespace="##any" processContents="lax"/>
3525 </complexType>
3526
3527 <element name="composite" type="sca:Composite"/>
3528 <complexType name="Composite">
3529     <sequence>
3530         <element name="include" type="anyURI" minOccurs="0"
3531             maxOccurs="unbounded"/>
3532         <choice minOccurs="0" maxOccurs="unbounded">
3533             <element name="service" type="sca:Service"/>
3534             <element name="property" type="sca:Property"/>
3535             <element name="component" type="sca:Component"/>
3536             <element name="reference" type="sca:Reference"/>
3537             <element name="wire" type="sca:Wire"/>
3538         </choice>
3539         <any namespace="##other" processContents="lax" minOccurs="0"
3540             maxOccurs="unbounded"/>
3541     </sequence>
3542     <attribute name="name" type="NCName" use="required"/>
3543     <attribute name="targetNamespace" type="anyURI" use="required"/>
3544     <attribute name="local" type="boolean" use="optional"
3545 default="false"/>
3546     <attribute name="autowire" type="boolean" use="optional"
3547 default="false"/>
3548     <attribute name="constrainingType" type="QName" use="optional"/>
3549     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3550     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3551     <anyAttribute namespace="##any" processContents="lax"/>
3552 </complexType>
3553
3554 <complexType name="Service">
3555     <sequence>
3556         <element ref="sca:interface" minOccurs="0" maxOccurs="1" />
3557         <element name="operation" type="sca:Operation" minOccurs="0"

```

```

3558         maxOccurs="unbounded" />
3559     <choice minOccurs="0" maxOccurs="unbounded">
3560         <element ref="sca:binding" />
3561         <any namespace="##other" processContents="lax"
3562             minOccurs="0" maxOccurs="unbounded" />
3563     </choice>
3564     <element ref="sca:callback" minOccurs="0" maxOccurs="1" />
3565     <any namespace="##other" processContents="lax" minOccurs="0"
3566         maxOccurs="unbounded" />
3567 </sequence>
3568 <attribute name="name" type="NCName" use="required" />
3569 <attribute name="promote" type="anyURI" use="required" />
3570 <attribute name="requires" type="sca:listOfQNames" use="optional" />
3571 <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3572 <anyAttribute namespace="##any" processContents="lax" />
3573 </complexType>
3574
3575 <element name="interface" type="sca:Interface" abstract="true" />
3576 <complexType name="Interface" abstract="true"/>
3577
3578 <complexType name="Reference">
3579     <sequence>
3580         <element ref="sca:interface" minOccurs="0" maxOccurs="1" />
3581         <element name="operation" type="sca:Operation" minOccurs="0"
3582             maxOccurs="unbounded" />
3583         <choice minOccurs="0" maxOccurs="unbounded">
3584             <element ref="sca:binding" />
3585             <any namespace="##other" processContents="lax" />
3586         </choice>
3587         <element ref="sca:callback" minOccurs="0" maxOccurs="1" />
3588         <any namespace="##other" processContents="lax" minOccurs="0"
3589             maxOccurs="unbounded" />
3590     </sequence>
3591     <attribute name="name" type="NCName" use="required" />
3592     <attribute name="target" type="sca:listOfAnyURIs" use="optional"/>
3593     <attribute name="wiredByImpl" type="boolean" use="optional"
3594 default="false"/>
3595     <attribute name="multiplicity" type="sca:Multiplicity"
3596         use="optional" default="1..1" />
3597     <attribute name="promote" type="sca:listOfAnyURIs" use="required" />
3598     <attribute name="requires" type="sca:listOfQNames" use="optional" />
3599     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3600     <anyAttribute namespace="##any" processContents="lax" />

```

```

3601     </complexType>
3602
3603     <complexType name="SCAPropertyBase" mixed="true">
3604         <!-- mixed="true" to handle simple type -->
3605         <sequence>
3606             <any namespace="##any" processContents="lax" minOccurs="0"
3607                 maxOccurs="1" />
3608             <!-- NOT an extension point; This xsd:any exists to accept
3609                 the element-based or complex type property
3610                 i.e. no element-based extension point under "sca:property"
3611 -->
3612         </sequence>
3613     </complexType>
3614
3615     <!-- complex type for sca:property declaration -->
3616     <complexType name="Property" mixed="true">
3617         <complexContent>
3618             <extension base="sca:SCAPropertyBase">
3619                 <!-- extension defines the place to hold default value -->
3620                 <attribute name="name" type="NCName" use="required"/>
3621                 <attribute name="type" type="QName" use="optional"/>
3622                 <attribute name="element" type="QName" use="optional"/>
3623                 <attribute name="many" type="boolean" default="false"
3624                     use="optional"/>
3625                 <attribute name="mustSupply" type="boolean" default="false"
3626                     use="optional"/>
3627                 <anyAttribute namespace="##any" processContents="lax"/>
3628                 <!-- an extension point ; attribute-based only -->
3629             </extension>
3630         </complexContent>
3631     </complexType>
3632
3633     <complexType name="PropertyValue" mixed="true">
3634         <complexContent>
3635             <extension base="sca:SCAPropertyBase">
3636                 <attribute name="name" type="NCName" use="required"/>
3637                 <attribute name="type" type="QName" use="optional"/>
3638                 <attribute name="element" type="QName" use="optional"/>
3639                 <attribute name="many" type="boolean" default="false"
3640                     use="optional"/>
3641                 <attribute name="source" type="string" use="optional"/>
3642                 <attribute name="file" type="anyURI" use="optional"/>
3643                 <anyAttribute namespace="##any" processContents="lax"/>

```

```

3644         <!-- an extension point ; attribute-based only -->
3645         </extension>
3646     </complexContent>
3647 </complexType>
3648
3649 <element name="binding" type="sca:Binding" abstract="true"/>
3650 <complexType name="Binding" abstract="true">
3651     <sequence>
3652         <element name="operation" type="sca:Operation" minOccurs="0"
3653             maxOccurs="unbounded" />
3654     </sequence>
3655     <attribute name="uri" type="anyURI" use="optional"/>
3656     <attribute name="name" type="NCName" use="optional"/>
3657     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3658     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3659 </complexType>
3660
3661 <element name="bindingType" type="sca:BindingType"/>
3662 <complexType name="BindingType">
3663     <sequence minOccurs="0" maxOccurs="unbounded">
3664         <any namespace="##other" processContents="lax" />
3665     </sequence>
3666     <attribute name="type" type="QName" use="required"/>
3667     <attribute name="alwaysProvides" type="sca:listOfQNames"
3668 use="optional"/>
3669     <attribute name="mayProvide" type="sca:listOfQNames" use="optional"/>
3670     <anyAttribute namespace="##any" processContents="lax"/>
3671 </complexType>
3672
3673 <element name="callback" type="sca:Callback"/>
3674 <complexType name="Callback">
3675     <choice minOccurs="0" maxOccurs="unbounded">
3676         <element ref="sca:binding"/>
3677         <any namespace="##other" processContents="lax"/>
3678     </choice>
3679     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3680     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3681     <anyAttribute namespace="##any" processContents="lax"/>
3682 </complexType>
3683
3684 <complexType name="Component">
3685     <sequence>
3686         <element ref="sca:implementation" minOccurs="0" maxOccurs="1"/>

```

```

3687         <choice minOccurs="0" maxOccurs="unbounded">
3688             <element name="service" type="sca:ComponentService"/>
3689             <element name="reference" type="sca:ComponentReference"/>
3690             <element name="property" type="sca:PropertyValue" />
3691         </choice>
3692         <any namespace="##other" processContents="lax" minOccurs="0"
3693             maxOccurs="unbounded"/>
3694     </sequence>
3695     <attribute name="name" type="NCName" use="required"/>
3696     <attribute name="autowire" type="boolean" use="optional" />
3697     <attribute name="constrainingType" type="QName" use="optional"/>
3698     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3699     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3700     <anyAttribute namespace="##any" processContents="lax"/>
3701 </complexType>
3702
3703 <complexType name="ComponentService">
3704     <complexContent>
3705         <restriction base="sca:Service">
3706             <sequence>
3707                 <element ref="sca:interface" minOccurs="0"
3708 maxOccurs="1"/>
3709                 <element name="operation" type="sca:Operation"
3710 minOccurs="0"
3711                 maxOccurs="unbounded" />
3712                 <choice minOccurs="0" maxOccurs="unbounded">
3713                     <element ref="sca:binding"/>
3714                     <any namespace="##other" processContents="lax"
3715                         minOccurs="0" maxOccurs="unbounded"/>
3716                 </choice>
3717                 <element ref="sca:callback" minOccurs="0"
3718 maxOccurs="1"/>
3719                 <any namespace="##other" processContents="lax"
3720 minOccurs="0"
3721                 maxOccurs="unbounded"/>
3722             </sequence>
3723             <attribute name="name" type="NCName" use="required"/>
3724             <attribute name="requires" type="sca:listOfQNames"
3725                 use="optional"/>
3726             <attribute name="policySets" type="sca:listOfQNames"
3727                 use="optional"/>
3728             <anyAttribute namespace="##any" processContents="lax"/>
3729         </restriction>
3730     </complexContent>

```

```

3731     </complexType>
3732
3733     <complexType name="ComponentReference">
3734         <complexContent>
3735             <restriction base="sca:Reference">
3736                 <sequence>
3737                     <element ref="sca:interface" minOccurs="0"
3738 maxOccurs="1" />
3739                     <element name="operation" type="sca:Operation"
3740 minOccurs="0"
3741                         maxOccurs="unbounded" />
3742                     <choice minOccurs="0" maxOccurs="unbounded">
3743                         <element ref="sca:binding" />
3744                         <any namespace="##other" processContents="lax"
3745 />
3746                     </choice>
3747                     <element ref="sca:callback" minOccurs="0"
3748 maxOccurs="1" />
3749                     <any namespace="##other" processContents="lax"
3750 minOccurs="0"
3751                         maxOccurs="unbounded" />
3752                 </sequence>
3753                 <attribute name="name" type="NCName" use="required" />
3754                 <attribute name="autowire" type="boolean" use="optional" />
3755                 <attribute name="wiredByImpl" type="boolean" use="optional"
3756                     default="false"/>
3757                 <attribute name="target" type="sca:listOfAnyURIs"
3758 use="optional"/>
3759                 <attribute name="multiplicity" type="sca:Multiplicity"
3760                     use="optional" default="1..1" />
3761                 <attribute name="requires" type="sca:listOfQNames"
3762 use="optional"/>
3763                 <attribute name="policySets" type="sca:listOfQNames"
3764                     use="optional"/>
3765                 <anyAttribute namespace="##any" processContents="lax" />
3766             </restriction>
3767         </complexContent>
3768     </complexType>
3769
3770     <element name="implementation" type="sca:Implementation"
3771         abstract="true" />
3772     <complexType name="Implementation" abstract="true">
3773         <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3774         <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3775     </complexType>

```



```

3776
3777     <element name="implementationType" type="sca:ImplementationType"/>
3778     <complexType name="ImplementationType">
3779         <sequence minOccurs="0" maxOccurs="unbounded">
3780             <any namespace="##other" processContents="lax" />
3781         </sequence>
3782         <attribute name="type" type="QName" use="required"/>
3783         <attribute name="alwaysProvides" type="sca:listOfQNames"
3784 use="optional"/>
3785         <attribute name="mayProvide" type="sca:listOfQNames" use="optional"/>
3786         <anyAttribute namespace="##any" processContents="lax"/>
3787     </complexType>
3788
3789     <complexType name="Wire">
3790         <sequence>
3791             <any namespace="##other" processContents="lax" minOccurs="0"
3792 maxOccurs="unbounded"/>
3793         </sequence>
3794         <attribute name="source" type="anyURI" use="required"/>
3795         <attribute name="target" type="anyURI" use="required"/>
3796         <anyAttribute namespace="##any" processContents="lax"/>
3797     </complexType>
3798
3799     <element name="include" type="sca:Include"/>
3800     <complexType name="Include">
3801         <attribute name="name" type="QName"/>
3802         <anyAttribute namespace="##any" processContents="lax"/>
3803     </complexType>
3804
3805     <complexType name="Operation">
3806         <attribute name="name" type="NCName" use="required"/>
3807         <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3808         <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3809         <anyAttribute namespace="##any" processContents="lax"/>
3810     </complexType>
3811
3812     <element name="constrainingType" type="sca:ConstrainingType"/>
3813     <complexType name="ConstrainingType">
3814         <sequence>
3815             <choice minOccurs="0" maxOccurs="unbounded">
3816                 <element name="service" type="sca:ComponentService"/>
3817                 <element name="reference" type="sca:ComponentReference"/>
3818                 <element name="property" type="sca:Property" />

```

```

3819         </choice>
3820         <any namespace="##other" processContents="lax" minOccurs="0"
3821             maxOccurs="unbounded" />
3822     </sequence>
3823     <attribute name="name" type="NCName" use="required" />
3824     <attribute name="targetNamespace" type="anyURI" />
3825     <attribute name="requires" type="sca:listOfQNames" use="optional" />
3826     <anyAttribute namespace="##any" processContents="lax" />
3827 </complexType>
3828
3829
3830 <simpleType name="Multiplicity">
3831     <restriction base="string">
3832         <enumeration value="0..1" />
3833         <enumeration value="1..1" />
3834         <enumeration value="0..n" />
3835         <enumeration value="1..n" />
3836     </restriction>
3837 </simpleType>
3838
3839 <simpleType name="OverrideOptions">
3840     <restriction base="string">
3841         <enumeration value="no" />
3842         <enumeration value="may" />
3843         <enumeration value="must" />
3844     </restriction>
3845 </simpleType>
3846
3847 <!-- Global attribute definition for @requires to permit use of intents
3848     within WSDL documents -->
3849 <attribute name="requires" type="sca:listOfQNames" />
3850
3851 <!-- Global attribute defintion for @endsConversation to mark operations
3852     as ending a conversation -->
3853 <attribute name="endsConversation" type="boolean" default="false" />
3854
3855 <simpleType name="listOfQNames">
3856     <list itemType="QName" />
3857 </simpleType>
3858
3859 <simpleType name="listOfAnyURIs">
3860     <list itemType="anyURI" />
3861 </simpleType>

```

3862
3863 </schema>

3864 **B.3 sca-binding-sca.xsd**

```
3865
3866 <?xml version="1.0" encoding="UTF-8"?>
3867 <!-- (c) Copyright SCA Collaboration 2006, 2007 -->
3868 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3869     targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3870     xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3871     elementFormDefault="qualified">
3872
3873     <include schemaLocation="sca-core.xsd"/>
3874
3875     <element name="binding.sca" type="sca:SCABinding"
3876         substitutionGroup="sca:binding"/>
3877     <complexType name="SCABinding">
3878         <complexContent>
3879             <extension base="sca:Binding">
3880                 <sequence>
3881                     <element name="operation" type="sca:Operation"
3882 minOccurs="0"
3883                         maxOccurs="unbounded" />
3884                 </sequence>
3885                 <attribute name="uri" type="anyURI" use="optional"/>
3886                 <attribute name="name" type="QName" use="optional"/>
3887                 <attribute name="requires" type="sca:listOfQNames"
3888                     use="optional"/>
3889                 <attribute name="policySets" type="sca:listOfQNames"
3890                     use="optional"/>
3891                 <anyAttribute namespace="##any" processContents="lax"/>
3892             </extension>
3893         </complexContent>
3894     </complexType>
3895 </schema>
```

3896

3897 **B.4 sca-interface-java.xsd**

3898

```
3899 <?xml version="1.0" encoding="UTF-8"?>
3900 <!-- (c) Copyright SCA Collaboration 2006 -->
3901 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3902     targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
```

```

3903     xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3904         elementFormDefault="qualified">
3905
3906     <include schemaLocation="sca-core.xsd"/>
3907
3908     <element name="interface.java" type="sca:JavaInterface"
3909         substitutionGroup="sca:interface"/>
3910     <complexType name="JavaInterface">
3911         <complexContent>
3912             <extension base="sca:Interface">
3913                 <sequence>
3914                     <any namespace="##other" processContents="lax"
3915 minOccurs="0"                                maxOccurs="unbounded"/>
3916                 </sequence>
3917                 <attribute name="interface" type="NCName" use="required"/>
3918                 <attribute name="callbackInterface" type="NCName"
3919 use="optional"/>
3920                 <anyAttribute namespace="##any" processContents="lax"/>
3921             </extension>
3922         </complexContent>
3923     </complexType>
3924 </schema>
3925

```

3926 B.5 sca-interface-wsdl.xsd

```

3927
3928 <?xml version="1.0" encoding="UTF-8"?>
3929 <!-- (c) Copyright SCA Collaboration 2006 -->
3930 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3931     targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3932     xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3933         elementFormDefault="qualified">
3934
3935     <include schemaLocation="sca-core.xsd"/>
3936
3937     <element name="interface.wsdl" type="sca:WSDLPortType"
3938         substitutionGroup="sca:interface"/>
3939     <complexType name="WSDLPortType">
3940         <complexContent>
3941             <extension base="sca:Interface">
3942                 <sequence>
3943                     <any namespace="##other" processContents="lax"
3944 minOccurs="0"                                maxOccurs="unbounded"/>
3945                 </sequence>
3946                 <attribute name="interface" type="anyURI" use="required"/>

```

```

3947         <attribute name="callbackInterface" type="anyURI"
3948 use="optional"/>
3949         <anyAttribute namespace="##any" processContents="lax"/>
3950     </extension>
3951 </complexContent>
3952 </complexType>
3953 </schema>
3954

```

3955 **B.6 sca-implementation-java.xsd**

```

3956
3957 <?xml version="1.0" encoding="UTF-8"?>
3958 <!-- (c) Copyright SCA Collaboration 2006 -->
3959 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3960     targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3961     xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3962     elementFormDefault="qualified">
3963
3964     <include schemaLocation="sca-core.xsd"/>
3965
3966     <element name="implementation.java" type="sca:JavaImplementation"
3967         substitutionGroup="sca:implementation"/>
3968     <complexType name="JavaImplementation">
3969         <complexContent>
3970             <extension base="sca:Implementation">
3971                 <sequence>
3972                     <any namespace="##other" processContents="lax"
3973                         minOccurs="0" maxOccurs="unbounded"/>
3974                 </sequence>
3975                 <attribute name="class" type="NCName" use="required"/>
3976                 <attribute name="requires" type="sca:listOfQNames"
3977 use="optional"/>
3978                 <attribute name="policySets" type="sca:listOfQNames"
3979                     use="optional"/>
3980                 <anyAttribute namespace="##any" processContents="lax"/>
3981             </extension>
3982         </complexContent>
3983     </complexType>
3984 </schema>

```

3985 **B.7 sca-implementation-composite.xsd**

```

3986
3987 <?xml version="1.0" encoding="UTF-8"?>

```

```

3988 <!-- (c) Copyright SCA Collaboration 2006 -->
3989 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3990     targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3991     xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3992     elementFormDefault="qualified">
3993
3994     <include schemaLocation="sca-core.xsd"/>
3995     <element name="implementation.composite" type="sca:SCAImplementation"
3996         substitutionGroup="sca:implementation"/>
3997     <complexType name="SCAImplementation">
3998         <complexContent>
3999             <extension base="sca:Implementation">
4000                 <sequence>
4001                     <any namespace="##other" processContents="lax"
4002 minOccurs="0"
4003                         maxOccurs="unbounded"/>
4004                 </sequence>
4005                 <attribute name="name" type="QName" use="required"/>
4006                 <attribute name="requires" type="sca:listOfQNames"
4007 use="optional"/>
4008                 <attribute name="policySets" type="sca:listOfQNames"
4009                     use="optional"/>
4010                 <anyAttribute namespace="##any" processContents="lax"/>
4011             </extension>
4012         </complexContent>
4013     </complexType>
4014 </schema>
4015

```

4016 B.8 sca-definitions.xsd

```

4017
4018 <?xml version="1.0" encoding="UTF-8"?>
4019 <!-- (c) Copyright SCA Collaboration 2006 -->
4020 <schema xmlns="http://www.w3.org/2001/XMLSchema"
4021     targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4022     xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4023     elementFormDefault="qualified">
4024
4025     <include schemaLocation="sca-core.xsd"/>
4026
4027     <element name="definitions">
4028         <complexType>
4029             <choice minOccurs="0" maxOccurs="unbounded">

```

```
4030     <element ref="sca:intent"/>
4031     <element ref="sca:policySet"/>
4032     <element ref="sca:binding"/>
4033     <element ref="sca:bindingType"/>
4034     <element ref="sca:implementationType"/>
4035     <any namespace="##other" processContents="lax" minOccurs="0"
4036         maxOccurs="unbounded"/>
4037     </choice>
4038 </complexType>
4039 </element>
4040
4041 </schema>
4042
```

4043 **B.9 sca-binding-webservice.xsd**

4044 Is described in [the SCA Web Services Binding specification \[9\]](#)

4045 **B.10 sca-binding-jms.xsd**

4046 Is described in [the SCA JMS Binding specification \[11\]](#)

4047 **B.11 sca-policy.xsd**

4048 Is described in [the SCA Policy Framework specification \[10\]](#)

4049

C. SCA Concepts

4050

C.1 Binding

4051 **Bindings** are used by services and references. References use bindings to describe the access
4052 mechanism used to call the service to which they are wired. Services use bindings to describe the
4053 access mechanism(s) that clients should use to call the service.

4054 SCA supports multiple different types of bindings. Examples include **SCA service, Web service,**
4055 **stateless session EJB, data base stored procedure, EIS service.** SCA provides an extensibility
4056 mechanism by which an SCA runtime can add support for additional binding types.

4057

4058

C.2 Component

4059 **SCA components** are configured instances of **SCA implementations**, which provide and consume
4060 services. SCA allows many different implementation technologies such as Java, BPEL, C++. SCA defines
4061 an **extensibility mechanism** that allows you to introduce new implementation types. The current
4062 specification does not mandate the implementation technologies to be supported by an SCA run-time,
4063 vendors may choose to support the ones that are important for them. A single SCA implementation may
4064 be used by multiple Components, each with a different configuration.

4065 The Component has a reference to an implementation of which it is an instance, a set of property values,
4066 and a set of service reference values. Property values define the values of the properties of the
4067 component as defined by the component's implementation. Reference values define the services that
4068 resolve the references of the component as defined by its implementation. These values can either be a
4069 particular service of a particular component, or a reference of the containing composite.

4070

C.3 Service

4071 **SCA services** are used to declare the externally accessible services of an **implementation**. For a
4072 composite, a service is typically provided by a service of a component within the composite, or by a
4073 reference defined by the composite. The latter case allows the republication of a service with a new
4074 address and/or new bindings. The service can be thought of as a point at which messages from external
4075 clients enter a composite or implementation.

4076 A service represents an addressable set of operations of an implementation that are designed to be
4077 exposed for use by other implementations or exposed publicly for use elsewhere (eg public Web services
4078 for use by other organizations). The operations provided by a service are specified by an Interface, as
4079 are the operations required by the service client (if there is one). An implementation may contain
4080 multiple services, when it is possible to address the services of the implementation separately.

4081 A service may be provided **as SCA remote services, as Web services, as stateless session EJB's, as**
4082 **EIS services, and so on.** Services use **bindings** to describe the way in which they are published. SCA
4083 provides an **extensibility mechanism** that makes it possible to introduce new binding types for new
4084 types of services.

4085

C.3.1 Remotable Service

4086 A Remotable Service is a service that is designed to be published remotely in a loosely-coupled
4087 SOA architecture. For example, SCA services of SCA implementations can define
4088 implementations of industry-standard web services. Remotable services use pass-by-value
4089 semantics for parameters and returned results.

4090 A service is remotable if it is defined by a WSDL port type or if it defined by a Java interface
4091 marked with the @Remotable annotation.

4092 C.3.2 Local Service

4093 Local services are services that are designed to be only used “locally” by other implementations
4094 that are deployed concurrently in a tightly-coupled architecture within the same operating system
4095 process.

4096 Local services may rely on by-reference calling conventions, or may assume a very fine-grained
4097 interaction style that is incompatible with remote distribution. They may also use technology-
4098 specific data-types.

4099 Currently a service is local only if it defined by a Java interface not marked with the @Remotable
4100 annotation.

4101

4102 C.4 Reference

4103 **SCA references** represent a dependency that an implementation has on a service that is supplied by
4104 some other implementation, where the service to be used is specified through configuration. In other
4105 words, a reference is a service that an implementation may call during the execution of its business
4106 function. References are typed by an interface.

4107 For composites, composite references can be accessed by components within the composite like any
4108 service provided by a component within the composite. Composite references can be used as the targets
4109 of wires from component references when configuring Components.

4110 A composite reference can be used to access a service such as: an SCA service provided by another
4111 SCA composite, a Web service, a stateless session EJB, a data base stored procedure or an EIS service,
4112 and so on. References use **bindings** to describe the access method used to their services. SCA provides
4113 an **extensibility mechanism** that allows the introduction of new binding types to references.

4114

4115 C.5 Implementation

4116 An implementation is concept that is used to describe a piece of software technology such as a Java
4117 class, BPEL process, XSLT transform, or C++ class that is used to implement one or more services in a
4118 service-oriented application. An SCA composite is also an implementation.

4119 Implementations define points of variability including properties that can be set and settable references to
4120 other services. The points of variability are configured by a component that uses the implementation. The
4121 specification refers to the configurable aspects of an implementation as its **componentType**.

4122 C.6 Interface

4123 **Interfaces** define one or more business functions. These business functions are provided by Services
4124 and are used by components through References. Services are defined by the Interface they implement.
4125 SCA currently supports two interface type systems:

- 4126 • Java interfaces
- 4127 • WSDL portTypes

4128

4129 SCA also provides an extensibility mechanism by which an SCA runtime can add support for additional
4130 interface type systems.

4131 Interfaces may be **bi-directional**. A bi-directional service has service operations which must be provided
4132 by each end of a service communication – this could be the case where a particular service requires a
4133 “callback” interface on the client, which is calls during the process of handing service requests from the
4134 client.

4135

4136 C.7 Composite

4137 An SCA composite is the basic unit of composition within an SCA Domain. An **SCA Composite** is an
4138 assembly of Components, Services, References, and the Wires that interconnect them. Composites can
4139 be used to contribute elements to an **SCA Domain**.

4140 A **composite** has the following characteristics:

- 4141 • It may be used as a component implementation. When used in this way, it defines a boundary for
4142 Component visibility. Components may not be directly referenced from outside of the composite
4143 in which they are declared.
- 4144 • It can be used to define a unit of deployment. Composites are used to contribute business logic
4145 artifacts to an SCA domain.

4146

4147 C.8 Composite inclusion

4148 One composite can be used to provide part of the definition of another composite, through the process of
4149 inclusion. This is intended to make team development of large composites easier. Included composites
4150 are merged together into the using composite at deployment time to form a single logical composite.

4151 Composites are included into other composites through <include.../> elements in the using composite.
4152 The SCA Domain uses composites in a similar way, through the deployment of composite files to a
4153 specific location.

4154

4155 C.9 Property

4156 **Properties** allow for the configuration of an implementation with externally set data values. The data
4157 value is provided through a Component, possibly sourced from the property of a containing composite.

4158 Each Property is defined by the implementation. Properties may be defined directly through the
4159 implementation language or through annotations of implementations, where the implementation language
4160 permits, or through a componentType file. A Property can be either a simple data type or a complex data
4161 type. For complex data types, XML schema is the preferred technology for defining the data types.

4162

4163 C.10 Domain

4164 An SCA Domain represents a set of Services providing an area of Business functionality that is controlled
4165 by a single organization. As an example, for the accounts department in a business, the SCA Domain
4166 might cover all finance-related functions, and it might contain a series of composites dealing with specific
4167 areas of accounting, with one for Customer accounts, another dealing with Accounts Payable.

4168 A domain specifies the instantiation, configuration and connection of a set of components, provided via
4169 one or more composite files. The domain, like a composite, also has Services and References. Domains
4170 also contain Wires which connect together the Components, Services and References.

4171

4172 C.11 Wire

4173 **SCA wires** connect **service references** to **services**.

4174 Within a composite, valid wire sources are component references and composite services. Valid wire
4175 targets are component services and composite references.

4176 When using included composites, the sources and targets of the wires don't have to be declared in the
4177 same composite as the composite that contains the wire. The sources and targets can be defined by
4178 other included composites. Targets can also be external to the SCA domain.

4179

4180

4181

4182 **D. Acknowledgements**

4183 The following individuals have participated in the creation of this specification and are gratefully
4184 acknowledged:

4185 **Participants:**

4186 [Participant Name, Affiliation | Individual Member]

4187 [Participant Name, Affiliation | Individual Member]

4188

E. Non-Normative Text

4190

F. Revision History

4191 [optional; should not be included in OASIS Standards]

4192

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

4193

4194

4195