



Service Component Architecture Assembly Model Specification Version 1.1

Working Draft

24 September, 2007

Specification URIs:

This Version:

<http://docs.oasis-open.org/sca-assembly/sca-assembly-draft-20070924.html>

<http://docs.oasis-open.org/sca-assembly/sca-assembly-draft-20070924.doc>

<http://docs.oasis-open.org/sca-assembly/sca-assembly-draft-20070924.pdf>

Previous Version:

Latest Version:

<http://docs.oasis-open.org/sca-assembly/sca-assembly-draft-20070924.html>

<http://docs.oasis-open.org/sca-assembly/sca-assembly-draft-20070924.doc>

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):

TBD

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® 2007. 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	Component.....	12
3.1	Example Component	16
4	Implementation.....	19
4.1	Component Type	20
4.1.1	Example ComponentType.....	22
4.1.2	Example Implementation.....	22
5	Interface.....	26
5.1	Local and Remotable Interfaces.....	27
5.2	Bidirectional Interfaces	28
5.3	Conversational Interfaces	28
5.4	SCA-Specific Aspects for WSDL Interfaces	31
6	Composite	33
6.1	Property – Definition and Configuration.....	36
6.1.1	Property Examples	37
6.2	References.....	41
6.2.1	Example Reference	43
6.3	Service	45
6.3.1	Service Examples.....	47
6.4	Wire.....	48
6.4.1	Wire Examples	50
6.4.2	Autowire.....	51
6.4.3	Autowire Examples.....	52
6.5	Using Composites as Component Implementations	55
6.5.1	Example of Composite used as a Component Implementation	57
6.6	Using Composites through Inclusion	58
6.6.1	Included Composite Examples.....	59
6.7	Composites which Include Component Implementations of Multiple Types	62
6.8	ConstrainingType.....	62
6.8.1	Example constrainingType.....	63
7	Binding.....	66
7.1	Messages containing Data not defined in the Service Interface	68
7.2	Form of the URI of a Deployed Binding.....	68
7.2.1	Constructing Hierarchical URIs	68
7.2.2	Non-hierarchical URIs	69
7.2.3	Determining the URI scheme of a deployed binding.....	69
7.3	SCA Binding	70
7.3.1	Example SCA Binding	70
7.4	Web Service Binding	71

7.5 JMS Binding.....	71
8 SCA Definitions	73
9 Extension Model.....	75
9.1 Defining an Interface Type.....	75
9.2 Defining an Implementation Type.....	77
9.3 Defining a Binding Type.....	78
10 Packaging and Deployment	82
10.1 Domains.....	82
10.2 Contributions.....	82
10.2.1 SCA Artifact Resolution.....	83
10.2.2 SCA Contribution Metadata Document.....	84
10.2.3 Contribution Packaging using ZIP.....	85
10.3 Installed Contribution	86
10.3.1 Installed Artifact URIs.....	86
10.4 Operations for Contributions.....	86
10.4.1 install Contribution & update Contribution.....	87
10.4.2 add Deployment Composite & update Deployment Composite.....	87
10.4.3 remove Contribution	87
10.5 Use of Existing (non-SCA) Mechanisms for Resolving Artifacts	87
10.6 Domain-Level Composite	88
10.6.1 add To Domain-Level Composite.....	88
10.6.2 remove From Domain-Level Composite	88
10.6.3 get Domain-Level Composite.....	88
10.6.4 get QName Definition	88
A. XML Schemas	90
A.1 sca.xsd	90
A.2 sca-core.xsd	90
A.3 sca-binding-sca.xsd.....	99
A.4 sca-interface-java.xsd	99
A.5 sca-interface-wsdl.xsd.....	100
A.6 sca-implementation-java.xsd.....	101
A.7 sca-implementation-composite.xsd.....	102
A.8 sca-definitions.xsd.....	102
A.9 sca-binding-webservice.xsd	103
A.10 sca-binding-jms.xsd.....	103
A.11 sca-policy.xsd.....	103
B. SCA Concepts.....	105
B.1 Binding.....	105
B.2 Component.....	105
B.3 Service.....	105
B.3.1 Remotable Service.....	105
B.3.2 Local Service.....	106
B.4 Reference	106
B.5 Implementation	106
B.6 Interface.....	106

B.7 Composite	107
B.8 Composite inclusion	107
B.9 Property	107
B.10 Domain	107
B.11 Wire	107
C. Acknowledgements	109
D. Non-Normative Text	110
E. Revision History.....	111

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]

57

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.

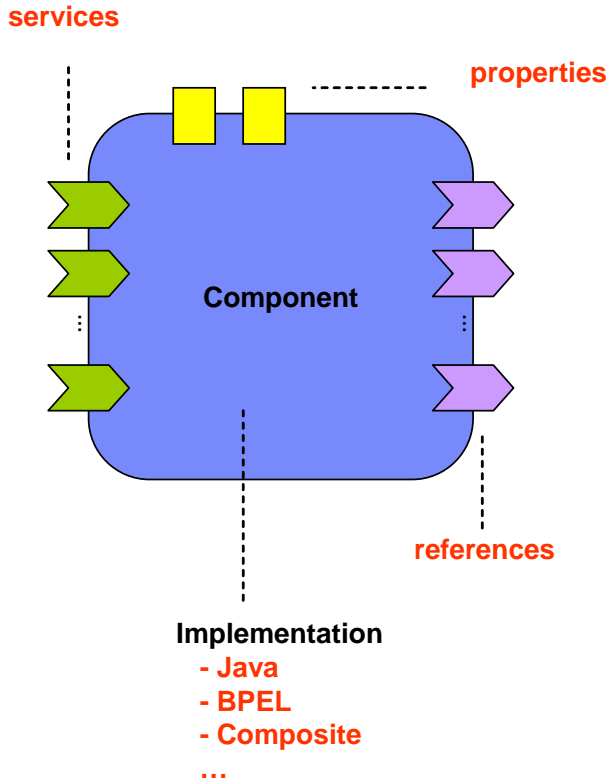
95 Composites are deployed within an **SCA Domain**. An SCA Domain typically represents a set of services
96 providing an area of business functionality that is controlled by a single organization. As an example, for
97 the accounts department in a business, the SCA Domain might cover all financial related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

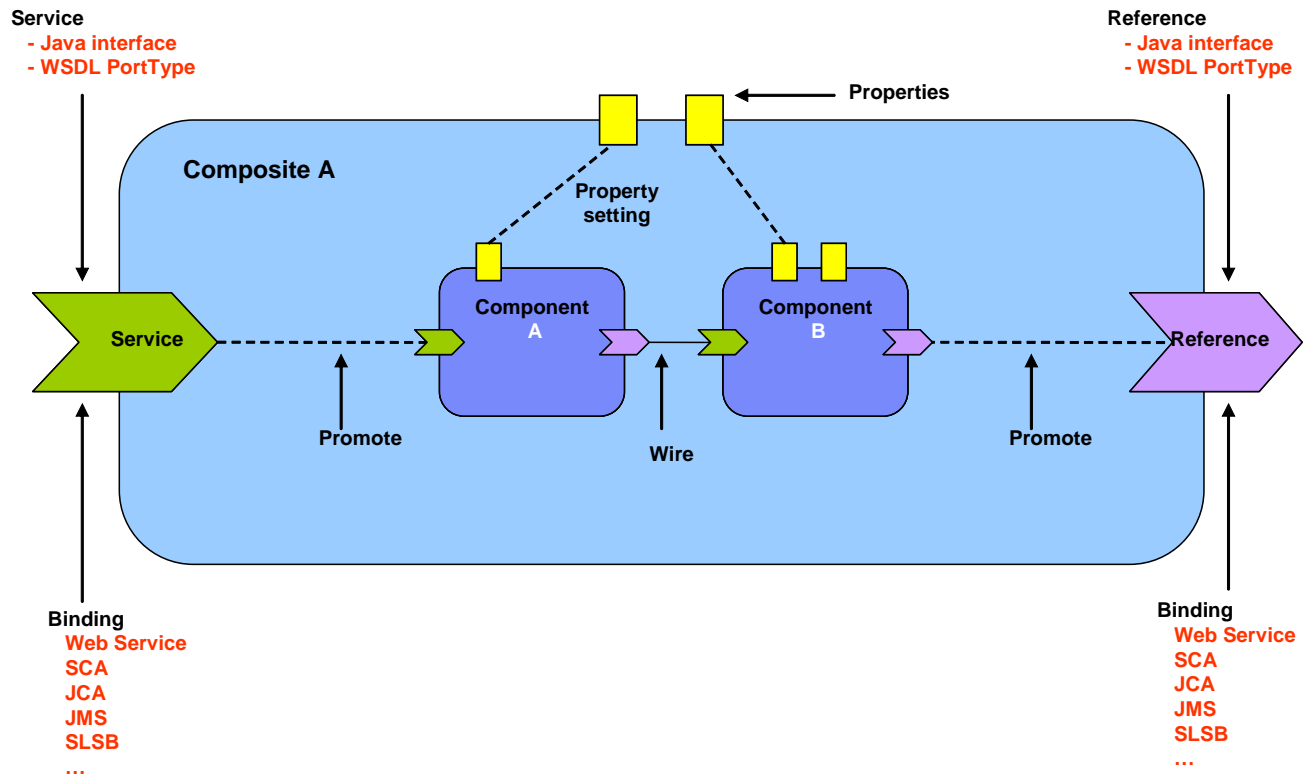
111 The following picture illustrates some of the features of an SCA component:



112
113 *Figure 1: SCA Component Diagram*

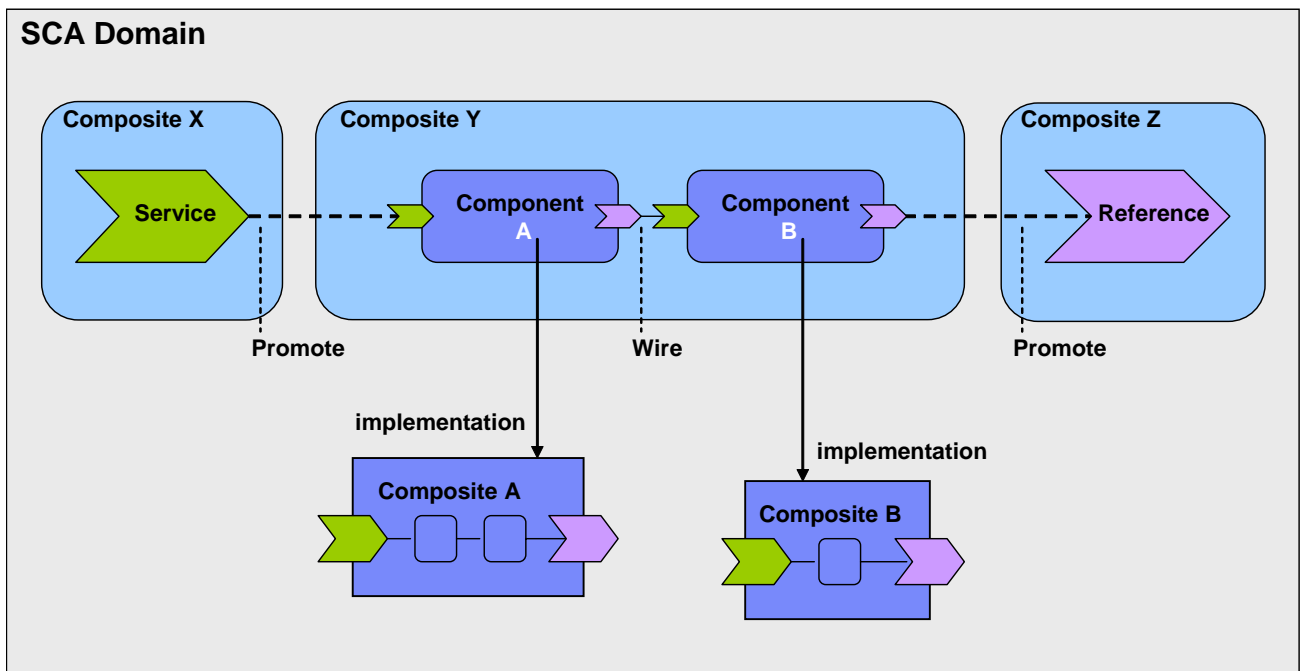
114
115 The following picture illustrates some of the features of a composite assembled using a set of
116 components:

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*

3 Component

125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166

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

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

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

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Component schema snippet -->
<composite xmlns="http://www.oesa.org/xmlns/sca/1.0"
  targetNamespace="xs:anyURI"
  name="xs:NCName" local="xs:boolean"?
  autowire="xs:boolean"? constrainingType="QName"?
  requires="list of xs:QName"? policySets="list of
xs:QName"?>
...
  <component name="xs:NCName" requires="list of xs:QName"?
    autowire="xs:boolean"?
    requires="list of xs:QName"? policySets="list of xs:QName"?
    constrainingType="xs:QName"?>*
    <implementation/>?
    <service name="xs:NCName" requires="list of xs:QName"?
      policySets="list of xs:QName"?>*
      <interface/>?
      <binding uri="xs:anyURI"? requires="list of xs:QName"?
        policySets="list of xs:QName"?/>*
    </service>
    <reference name="xs:NCName" multiplicity="0..1 or 1..1 or 0..n or
1..n"?
      autowire="xs:boolean"?
      target="list of xs:anyURI"? policySets="list of xs:QName"?
      wiredByImpl="xs:boolean"? requires="list of xs:QName"?>*
      <interface/>?
      <binding uri="xs:anyURI"? requires="list of xs:QName"?
        policySets="list of xs:QName"?/>*
    </reference>
```

```

167         <property name="xs:NCName" (type="xs:QName" |
168 element="xs:QName")?
169         mustSupply="xs:boolean"?
170         many="xs:boolean"? source="xs:string"? file="xs:anyURI"?>*
171         property-value?
172     </property>
173 </component>
174
175     ...
176
177 </composite>

```

178 The component element has the following *attributes*:

- 181 • **name (required)** – the name of the component. The name must be unique across all the
182 components in the composite.
- 183 • **autowire (optional)** – whether contained component references should be autowired, as
184 described in [the Autowire section](#). Default is false.
- 185 • **requires (optional)** – a list of policy intents. See the [Policy Framework specification \[10\]](#)
186 for a description of this attribute.
- 187 • **policySets (optional)** – a list of policy sets. See the [Policy Framework specification \[10\]](#)
188 for a description of this attribute.
- 189 • **constrainingType (optional)** – the name of a constrainingType. When specified, the set
190 of services, references and properties of the component, plus related intents, is
191 constrained to the set defined by the constrainingType. See [the ConstrainingType Section](#)
192 for more details.

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

198 The component element can have **zero or more service elements** as children which are used to
199 configure the services of the component. The services that can be configured are defined by the
200 implementation.

201

202 The service element has the following *attributes*:

- 203 • **name (required)** - the name of the service. Has to match a name of a service defined
204 by the implementation.
- 205 • **requires (optional)** – a list of policy intents. See the [Policy Framework specification \[10\]](#)
206 for a description of this attribute.
207 Note: The effective set of policy intents for the service consists of any intents explicitly
208 stated in this requires attribute, combined with any intents specified for the service by the
209 implementation.
- 210 • **policySets (optional)** – a list of policy sets. See the [Policy Framework specification \[10\]](#)
211 for a description of this attribute.

212

213 A service has **zero or one interface**, which describes the operations provided by the service. The
214 interface is described by an **interface element** which is a child element of the service element. If

215 no interface is specified, then the interface specified for the service by the implementation is in
216 effect. If an interface is specified it must provide a compatible subset of the interface provided by
217 the implementation, i.e. provide a subset of the operations defined by the implementation for the
218 service. For details on the interface element see [the Interface section](#).

219 A service element has one or more **binding elements** as children. If no bindings are specified,
220 then the bindings specified for the service by the implementation are in effect. If bindings are
221 specified, then those bindings override the bindings specified by the implementation. Details of the
222 binding element are described in [the Bindings section](#). The binding, combined with any PolicySets
223 in effect for the binding, must satisfy the set of policy intents for the service, as described in [the](#)
224 [Policy Framework specification \[10\]](#).

225

226 The component element can have **zero or more reference elements** as children which are used
227 to configure the references of the component. The references that can be configured are defined
228 by the implementation.

229

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

- 231 • **name (required)** – the name of the reference. Has to match a name of a reference
232 defined by the implementation.
- 233 • **autowire (optional)** – whether the reference should be autowired, as described in [the](#)
234 [Autowire section](#). Default is false.
- 235 • **requires (optional)** – a list of policy intents. See the [Policy Framework specification \[10\]](#)
236 for a description of this attribute.
237 Note: The effective set of policy intents for the reference consists of any intents explicitly
238 stated in this requires attribute, combined with any intents specified for the reference by
239 the implementation.
- 240 • **policySets (optional)** – a list of policy sets. See the [Policy Framework specification \[10\]](#)
241 for a description of this attribute.
- 242 • **multiplicity (optional)** - defines the number of wires that can connect the reference to
243 target services. Overrides the multiplicity specified for this reference on the
244 implementation. The value can only be equal or further restrict, i.e. 0..n to 0..1 or 1..n to
245 1..1. The multiplicity can have the following values
 - 246 ○ 1..1 – one wire can have the reference as a source
 - 247 ○ 0..1 – zero or one wire can have the reference as a source
 - 248 ○ 1..n – one or more wires can have the reference as a source
 - 249 ○ 0..n - zero or more wires can have the reference as a source
- 250 • **target (optional)** – a list of one or more of target service URI's, depending on multiplicity
251 setting. Each value wires the reference to a component service that resolves the
252 reference. For more details on wiring see [the section on Wires](#). Overrides any target
253 specified for this reference on the implementation.
- 254 • **wiredByImpl (optional)** – a boolean value, "false" by default, which indicates that the
255 implementation wires this reference dynamically. If set to "true" it indicates that the
256 target of the reference is set at runtime by the implementation code (eg by the code
257 obtaining an endpoint reference by some means and setting this as the target of the
258 reference through the use of programming interfaces defined by the relevant Client and
259 Implementation specification). If "true" is set, then the reference should not be wired
260 statically within a composite, but left unwired.

261

262 A reference has **zero or one interface**, which describes the operations required by the reference.
263 The interface is described by an **interface element** which is a child element of the reference
264 element. If no interface is specified, then the interface specified for the reference by the

265 implementation is in effect. If an interface is specified it must provide a compatible superset of the
266 interface provided by the implementation, i.e. provide a superset of the operations defined by the
267 implementation for the reference. For details on the interface element see [the Interface section](#).

268 A reference element has one or more **binding elements** as children. If no bindings are specified,
269 then the bindings specified for the reference by the implementation are in effect. If any bindings
270 are specified, then those bindings override any and all the bindings specified by the
271 implementation. Details of the binding element are described in the [Bindings section](#). The binding,
272 combined with any PolicySets in effect for the binding, must satisfy the set of policy intents for the
273 reference, as described in [the Policy Framework specification \[10\]](#).

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

281
282 The component element has **zero or more property elements** as its children, which are used to
283 configure data values of properties of the implementation. Each property element provides a value
284 for the named property, which is passed to the implementation. The properties that can be
285 configured and their types are defined by the implementation. An implementation can declare a
286 property as multi-valued, in which case, multiple property values can be present for a given
287 property.

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

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

292
293 The form of the value of the source attribute follows the form of an XPath expression.
294 This form allows a specific property of the composite to be addressed by name. Where the
295 property is complex, the XPath expression can be extended to refer to a sub-part of the
296 complex value.

297
298 So, for example, `source="$currency"` is used to reference a property of the composite
299 called "currency", while `source="$currency/a"` references the sub-part "a" of the
300 complex composite property with the name "currency".

- 301
- 302 • By specifying a dereferencable URI to a file containing the property value through the **file**
303 attribute. The contents of the referenced file are used as the value of the property.

304
305 If more than one property value specification is present, the source attribute takes precedence, then
306 the file attribute.

307
308 Optionally, the type of the property can be specified in **one** of two ways:

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

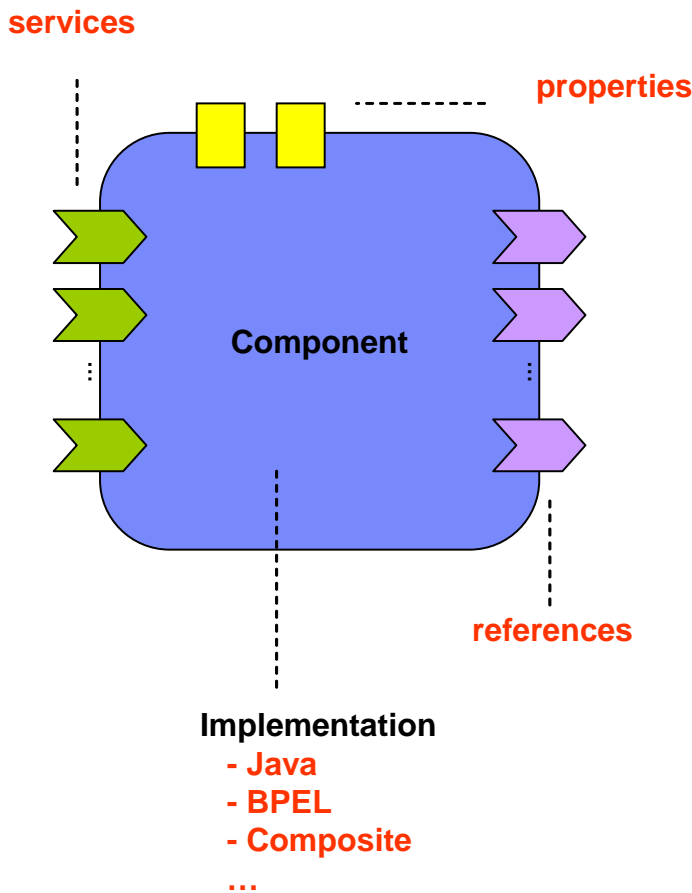
311 The property type specified must be compatible with the type of the property declared by the
312 implementation. If no type is specified, the type of the property declared by the implementation is
313 used.

314
315 The property element has the following attributes:

- 316 ▪ **name (required)** – the name of the property. Has to match a name of a property defined
- 317 by the implementation
- 318 ▪ **type (optional)** – the type of the property defined as the qualified name of an XML
- 319 schema type
- 320 ▪ **element (optional)** – the type of the property defined as the qualified name of an XML
- 321 schema global element – the type is the type of the global element
- 322 ▪ **source (optional)** – an XPath expression pointing to a property of the containing
- 323 composite from which the value of this component property is obtained.
- 324 ▪ **file (optional)** – a dereferencable URI to a file containing a value for the property
- 325 ▪ **many (optional)** – (optional) whether the property is single-valued (false) or multi-
- 326 valued (true). Overrides the many specified for this property on the implementation. The
- 327 value can only be equal or further restrict, i.e. if the implementation specifies many true,
- 328 then the component can say false. In the case of a multi-valued property, it is presented
- 329 to the implementation as a Collection of property values.
- 330
- 331

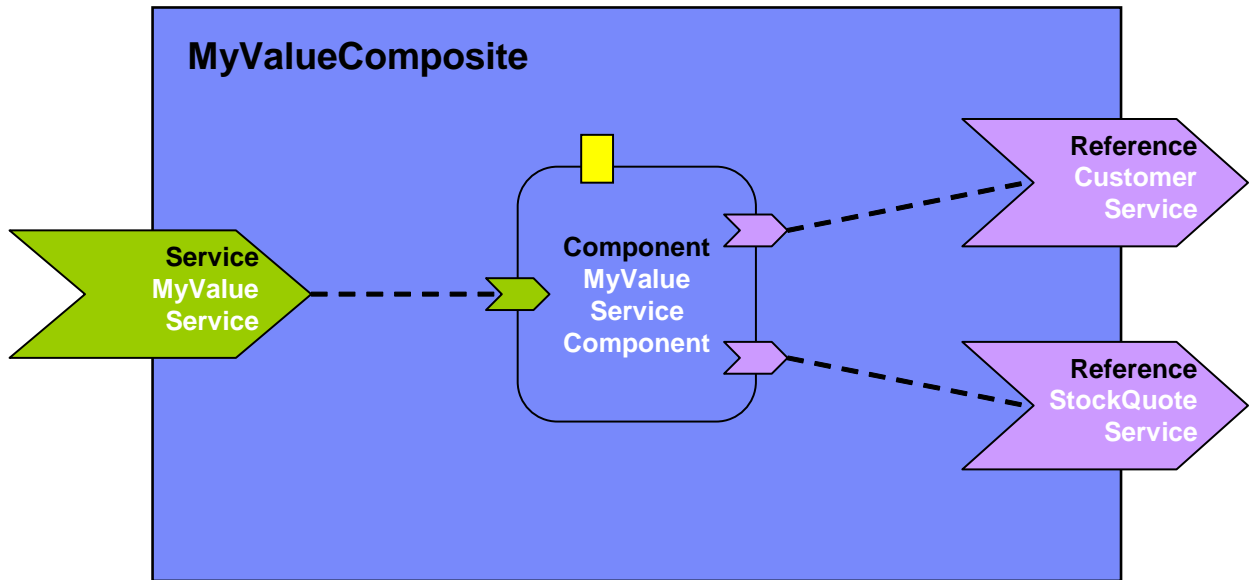
3.1 Example Component

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



336
337 *Figure 4: Component symbol*

338 The following figure shows the assembly diagram for the MyValueComposite containing the
 339 MyValueServiceComponent.
 340



341
 342
 343 *Figure 5: Assembly diagram for MyValueComposite*

344
 345 The following snippet shows the MyValueComposite.composite file for the MyValueComposite
 346 containing the component element for the MyValueServiceComponent. A value is set for the
 347 property named currency, and the customerService and stockQuoteService references are
 348 promoted:

```

349
350 <?xml version="1.0" encoding="ASCII"?>
351 <!-- MyValueComposite_1 example -->
352 <composite xmlns="http://www.oesa.org/xmlns/sca/1.0"
353           targetNamespace="http://foo.com"
354           name="MyValueComposite" >
355
356     <service name="MyValueService" promote="MyValueServiceComponent"/>
357
358     <component name="MyValueServiceComponent">
359       <implementation.java
360 class="services.myvalue.MyValueServiceImpl"/>
361       <property name="currency">EURO</property>
362       <reference name="customerService"/>
363       <reference name="stockQuoteService"/>
364     </component>
365
366     <reference name="CustomerService"
367               promote="MyValueServiceComponent/customerService"/>
  
```

```
368
369     <reference name="StockQuoteService"
370         promote="MyValueServiceComponent/stockQuoteService"/>
371
372 </composite>
```

373
374 Note that the references of MyValueServiceComponent are explicitly declared only for purposes of
375 clarity – the references are defined by the MyValueServiceImpl implementation and there is no
376 need to redeclare them on the component unless the intention is to wire them or to override some
377 aspect of them.

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

```
381 <?xml version="1.0" encoding="ASCII"?>
382 <!-- MyValueComposite_2 example -->
383 <composite xmlns="http://www.oxa.org/xmlns/sca/1.0"
384     targetNamespace="http://foo.com"
385     name="MyValueComposite" >
386
387     <service name="MyValueService" promote="MyValueServiceComponent"/>
388
389     <component name="MyValueServiceComponent">
390         <implementation.java
391 class="services.myvalue.MyValueServiceImpl"/>
392         <property name="currency">EURO</property>
393         <property name="currency">Yen</property>
394         <property name="currency">USDollar</property>
395         <reference name="customerService"
396             target="InternalCustomer/customerService"/>
397         <reference name="StockQuoteService"/>
398     </component>
399
400     ...
401
402     <reference name="CustomerService"
403         promote="MyValueServiceComponent/customerService"/>
404
405     <reference name="StockQuoteService"
406         promote="MyValueServiceComponent/StockQuoteService"/>
407
408 </composite>
```

409
410this assumes that the composite has another component called InternalCustomer (not shown)
411 which has a service to which the customerService reference of the MyValueServiceComponent is
412 wired as well as being promoted externally through the composite reference CustomerService.

413 4 Implementation

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

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

422 For example, within a component declaration in a composite file, the elements
423 **implementation.java** and **implementation.bpel** point to Java and BPEL implementation types
424 respectively. **implementation.composite** points to the use of an SCA composite as an
425 implementation. **implementation.spring** and **implementation.ejb** are used for Java
426 components written to the Spring framework and the Java EE EJB technology respectively.

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

429

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

431

```
432 <implementation.bpel process="MoneyTransferProcess"/>
```

433

```
434 <implementation.composite name="MyValueComposite"/>
```

435

436 **Services, references and properties** are the configurable aspects of an implementation. SCA
437 refers to them collectively as the **component type**. The characteristics of services, references and
438 properties are described in the Component section. Depending on the implementation type, the
439 implementation may be able to declare the services, references and properties that it has and it
440 also may be able to set values for all the characteristics of those services, references and
441 properties.

442 So, for example:

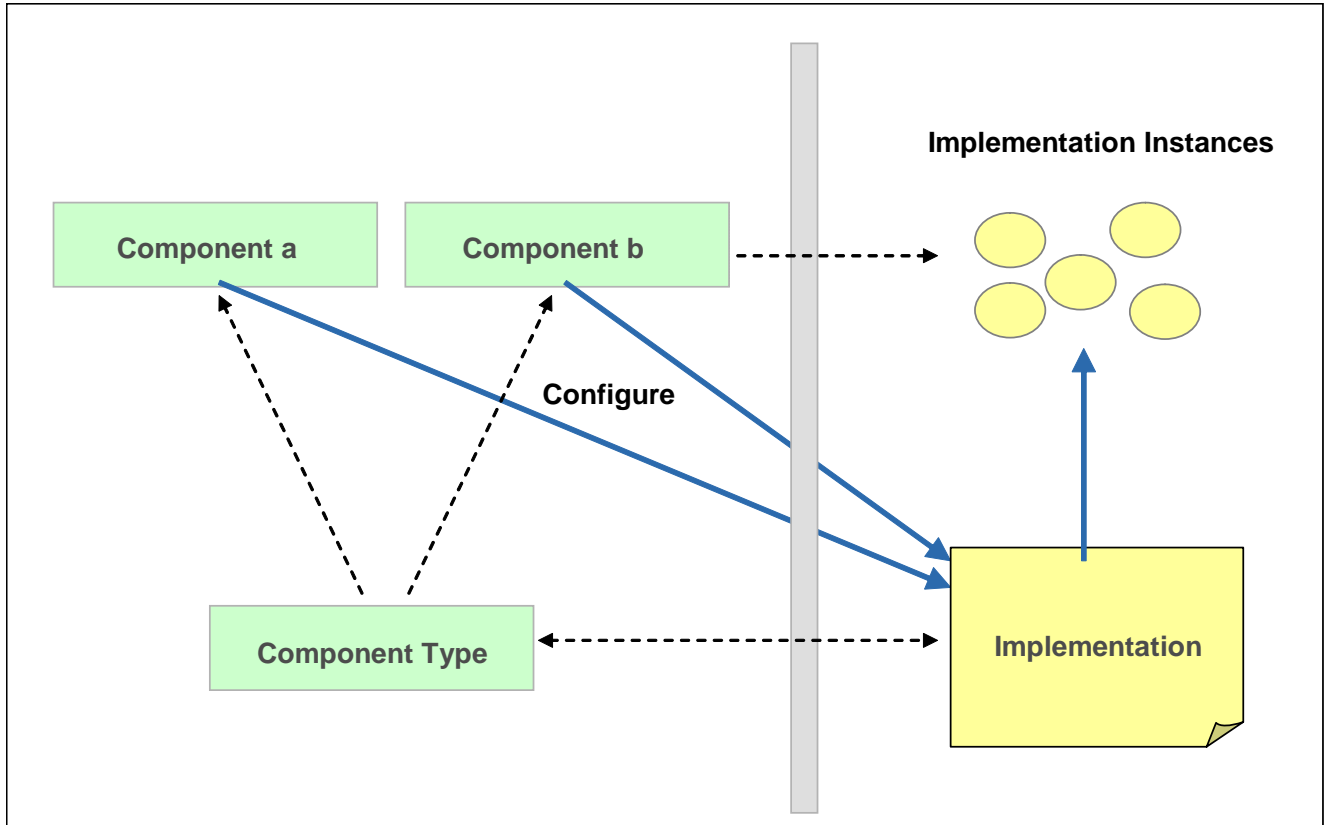
- 443 • for a service, the implementation may define the interface, binding(s), a URI, intents, and
444 policy sets, including details of the bindings
- 445 • for a reference, the implementation may define the interface, binding(s), target URI(s),
446 intents, policy sets, including details of the bindings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

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

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

457 At runtime, an **implementation instance** is a specific runtime instantiation of the
458 implementation – its runtime form depends on the implementation technology used. The
459 implementation instance derives its business logic from the implementation on which it is based,

460 but the values for its properties and references are derived from the component which configures
461 the implementation.



462
463 *Figure 6: Relationship of Component and Implementation*

464
465 **4.1 Component Type**

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

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

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

482 A **component type file** has the same name as the implementation file but has the extension
483 ".**componentType**". The component type is defined by a **componentType element** in the file.
484 The **location** of the component type file depends on the type of the component implementation: it
485 is described in the respective client and implementation model specification for the implementation
486 type.

487 The componentType element can contain Service elements, Reference elements and Property
488 elements.

489 The following snippet shows the componentType schema.

490

```
491 <?xml version="1.0" encoding="ASCII"?>
492 <!-- Component type schema snippet -->
493 <componentType xmlns="http://www.oesa.org/xmlns/sca/1.0"
494     constrainingType="QName"? >
495
496     <service name="xs:NCName" requires="list of xs:QName"?
497         policySets="list of xs:QName"?>*
498         <interface/>
499         <binding uri="xs:anyURI"? requires="list of xs:QName"?
500             policySets="list of xs:QName"?/>*
501     </service>
502
503     <reference name="xs:NCName" target="list of xs:anyURI"?
504         multiplicity="0..1 or 1..1 or 0..n or 1..n"?
505         wiredByImpl="xs:boolean"? requires="list of xs:QName"?
506         policySets="list of xs:QName"?>*
507     <interface/>?
508     <binding uri="xs:anyURI"? requires="list of xs:QName"?
509         policySets="list of xs:QName"?/>*
510 </reference>
511
512     <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
513         many="xs:boolean"? mustSupply="xs:boolean"?
514         policySets="list of xs:QName"?>*
515         default-property-value?
516     </property>
517
518     <implementation requires="list of xs:QName"?
519         policySets="list of xs:QName"?/>?
520
521 </componentType>
522
```

523 ComponentType has a single attribute:

- 524 • **constrainingType (optional)** – the name of a constrainingType. When specified, the set
525 of services, references and properties of the implementation, plus related intents, is
526 constrained to the set defined by the constrainingType. See [the ConstrainingType Section](#)
527 for more details.

528 **A Service** represents an addressable interface of the implementation. The service is represented
529 by a **service element** which is a child of the componentType element. There can be **zero or**
530 **more** service elements in a componentType. See [the Service section](#) for details.

531
532 A **Reference** represents a requirement that the implementation has on a service provided by
533 another component. The reference is represented by a **reference element** which is a child of the
534 componentType element. There can be **zero or more** reference elements in a component type
535 definition. See [the Reference section](#) for details.

536
537 **Properties** allow for the configuration of an implementation with externally set values. Each
538 Property is defined as a property element. The componentType element can have zero or more
539 property elements as its children. See [the Property section](#) for details.

540
541 **Implementation** represents characteristics inherent to the implementation itself, in particular
542 intents and policies. See the [Policy Framework specification \[10\]](#) for a description of intents and
543 policies.

544

545 4.1.1 Example ComponentType

546

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

550

```
551 <?xml version="1.0" encoding="ASCII"?>
552 <componentType xmlns="http://www.osea.org/xmlns/sca/1.0">
553
554     <service name="MyValueService">
555         <interface.java interface="services.myvalue.MyValueService"/>
556     </service>
557
558     <reference name="customerService">
559         <interface.java interface="services.customer.CustomerService"/>
560     </reference>
561     <reference name="stockQuoteService">
562         <interface.java
563 interface="services.stockquote.StockQuoteService"/>
564     </reference>
565
566     <property name="currency" type="xsd:string">USD</property>
567
568 </componentType>
569
```

570 4.1.2 Example Implementation

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

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

```

575
576 package services.account;
577
578 @Remotable
579 public interface AccountService{
580
581     public AccountReport getAccountReport(String customerID);
582 }

```

583

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

587

```

588 package services.account;
589
590 import java.util.List;
591
592 import commonj.sdo.DataFactory;
593
594 import org.osoa.sca.annotations.Property;
595 import org.osoa.sca.annotations.Reference;
596
597 import services.accountdata.AccountDataService;
598 import services.accountdata.CheckingAccount;
599 import services.accountdata.SavingsAccount;
600 import services.accountdata.StockAccount;
601 import services.stockquote.StockQuoteService;
602
603 public class AccountServiceImpl implements AccountService {
604
605     @Property
606     private String currency = "USD";
607
608     @Reference
609     private AccountDataService accountDataService;
610     @Reference
611     private StockQuoteService stockQuoteService;
612
613     public AccountReport getAccountReport(String customerID) {
614
615         DataFactory dataFactory = DataFactory.INSTANCE;
616         AccountReport accountReport = (AccountReport)dataFactory.create(AccountReport.class);
617         List accountSummaries = accountReport.getAccountSummaries();
618
619         CheckingAccount checkingAccount = accountDataService.getCheckingAccount(customerID);
620         AccountSummary checkingAccountSummary =
621 (AccountSummary)dataFactory.create(AccountSummary.class);
622         checkingAccountSummary.setAccountNumber(checkingAccount.getAccountNumber());
623         checkingAccountSummary.setAccountType("checking");
624         checkingAccountSummary.setBalance(fromUSDollarToCurrency(checkingAccount.getBalance()));

```

```

625     accountSummaries.add(checkingAccountSummary);
626
627     SavingsAccount savingsAccount = accountDataService.getSavingsAccount(customerID);
628     AccountSummary savingsAccountSummary =
629 (AccountSummary)dataFactory.create(AccountSummary.class);
630     savingsAccountSummary.setAccountNumber(savingsAccount.getAccountNumber());
631     savingsAccountSummary.setAccountType("savings");
632     savingsAccountSummary.setBalance(fromUSDollarToCurrency(savingsAccount.getBalance()));
633     accountSummaries.add(savingsAccountSummary);
634
635     StockAccount stockAccount = accountDataService.getStockAccount(customerID);
636     AccountSummary stockAccountSummary =
637 (AccountSummary)dataFactory.create(AccountSummary.class);
638     stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());
639     stockAccountSummary.setAccountType("stock");
640     float balance=
641 (stockQuoteService.getQuote(stockAccount.getSymbol()))*stockAccount.getQuantity();
642     stockAccountSummary.setBalance(fromUSDollarToCurrency(balance));
643     accountSummaries.add(stockAccountSummary);
644
645     return accountReport;
646 }
647
648 private float fromUSDollarToCurrency(float value){
649
650     if (currency.equals("USD")) return value; else
651     if (currency.equals("EURO")) return value * 0.8f; else
652     return 0.0f;
653 }
654 }
655

```

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

```

658
659 <?xml version="1.0" encoding="ASCII"?>
660 <componentType xmlns="http://www.osea.org/xmlns/sca/1.0"
661     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
662
663     <service name="AccountService">
664         <interface.java interface="services.account.AccountService"/>
665     </service>
666     <reference name="accountDataService">
667         <interface.java
668 interface="services.accountdata.AccountDataService"/>
669     </reference>
670     <reference name="stockQuoteService">
671         <interface.java
672 interface="services.stockquote.StockQuoteService"/>
673     </reference>

```


674
675
676
677
678
679
680
681
682

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

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

683 5 Interface

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

690 SCA currently supports the following interface type systems:

- 691 • Java interfaces
- 692 • WSDL 1.1 portTypes
- 693 • WSDL 2.0 interfaces

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

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

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

698

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

700

701 The interface.java element has the following attributes:

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

703

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

705

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

707

708 Here, the Java interface is defined in the Java class file
709 ./services/stockquote/StockQuoteService.class, where the root directory is defined by the
710 contribution in which the interface exists.

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

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

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

718

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

720

721 The interface.wsdl element has the following attributes:

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

724

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

726

```
727 <interface wsdl:interface="http://www.stockquote.org/StockQuoteService#
728                                     wsdl:interface(StockQuo
729                                     te)"/>
```

730

731 For WSDL 1.1, the interface attribute points to a portType in the WSDL. For WSDL 2.0, the
732 interface attribute points to an interface in the WSDL. For the WSDL 1.1 portType and WSDL 2.0
733 interface type systems, arguments and return of the service operations are described using XML
734 schema.

735

736

737 5.1 Local and Remotable Interfaces

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

744

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

748

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

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

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

758

```
759 package services.hello;
760
761 @Remotable
762 public interface HelloService {
763
764     String hello(String message);
765 }
```

766

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

772

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

777

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

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

781

782 5.2 Bidirectional Interfaces

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

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

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

795

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

798

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

802

803 If a reference is defined using a bidirectional interface element, the client component
804 implementation using the reference calls the referenced service using the interface. The client
805 component implementation must implement the callback interface.

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

809 Facilities are provided within SCA which allow a different component to provide a callback interface
810 than the component which was the client to an original service invocation. How this is done can be
811 seen in the [SCA Java Client and Implementation Specification](#) (section named "Passing
812 Conversational Services as Parameters").

813

814 5.3 Conversational Interfaces

815

816 Services sometimes cannot easily be defined so that each operation stands alone and is
817 completely independent of the other operations of the same service. Instead, there is a sequence
818 of operations that must be called in order to achieve some higher level goal. SCA calls this

819 sequence of operations a **conversation**. If the service uses a bidirectional interface, the
820 conversation may include both operations and callbacks.

821
822 Such conversational services are typically managed by using conversation identifiers that are
823 either (1) part of the application data (message parts or operation parameters) or 2)
824 communicated separately from application data (possibly in headers). SCA introduces the concept
825 of *conversational interfaces* for describing the interface contract for conversational services of the
826 second form above. With this form, it is possible for the runtime to automatically manage the
827 conversation, with the help of an appropriate binding specified at deployment. SCA does not
828 standardize any aspect of conversational services that are maintained using application data.
829 Such services are neither helped nor hindered by SCA's conversational service support.

830
831 Conversational services typically involve state data that relates to the conversation that is taking
832 place. The creation and management of the state data for a conversation has a significant impact
833 on the development of both clients and implementations of conversational services.

834
835 Traditionally, application developers who have needed to write conversational services have been
836 required to write a lot of plumbing code. They need to:

- 837
- 838 - choose or define a protocol to communicate conversational (correlation) information
839 between the client & provider
 - 840 - route conversational messages in the provider to a machine that can handle that
841 conversation, while handling concurrent data access issues
 - 842 - write code in the client to use/encode the conversational information
 - 843 - maintain state that is specific to the conversation, sometimes persistently and
844 transactionally, both in the implementation and the client.

845
846 SCA makes it possible to divide the effort associated with conversational services between a
847 number of roles:

- 848 - Application Developer: Declares that a service interface is conversational (leaving the
849 details of the protocol up to the binding). Uses lifecycle semantics, APIs or other
850 programmatic mechanisms (as defined by the implementation-type being used) to
851 manage conversational state.
- 852 - Application Assembler: chooses a binding that can support conversations
- 853 - Binding Provider: implements a protocol that can pass conversational information with
854 each operation request/response.
- 855 - Implementation-Type Provider: defines APIs and/or other programmatic mechanisms for
856 application developers to access conversational information. Optionally implements
857 instance lifecycle semantics that automatically manage implementation state based on
858 the binding's conversational information.

859
860 This specification requires interfaces to be marked as conversational by means of a policy intent
861 with the name **"conversational"**. The form of the marking of this intent depends on the
862 interface type. Note that it is also possible for a service or a reference to set the conversational
863 intent when using an interface which is not marked with the conversational intent. This can be
864 useful when reusing an existing interface definition that does not contain SCA information.

865 The meaning of the conversational intent is that both the client and the provider of the interface
866 may assume that messages (in either direction) will be handled as part of an ongoing conversation
867 without depending on identifying information in the body of the message (i.e. in parameters of the

868 operations). In effect, the conversation interface specifies a high-level abstract protocol that must
869 be satisfied by any actual binding/policy combination used by the service.

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

- 871 - Web service binding with a WS-RM policy
- 872 - Web service binding with a WS-Addressing policy
- 873 - Web service binding with a WS-Context policy
- 874 - JMS binding with a conversation policy that uses the JMS correlationID header

875

876 Conversations occur between one client and one target service. Consequently, requests originating
877 from one client to multiple target conversational services will result in multiple conversations. For
878 example, if a client A calls services B and C, both of which implement conversational interfaces,
879 two conversations result, one between A and B and another between A and C. Likewise, requests
880 flowing through multiple implementation instances will result in multiple conversations. For
881 example, a request flowing from A to B and then from B to C will involve two conversations (A and
882 B, B and C). In the previous example, if a request was then made from C to A, a third
883 conversation would result (and the implementation instance for A would be different from the one
884 making the original request).

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

892

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

900 A *sca:ConversationViolation* fault is thrown when one of the following errors occurs:

- 901 - A message is received for a particular conversation, after the conversation has ended
- 902 - The conversation identification is invalid (not unique, out of range, etc.)
- 903 - The conversation identification is not present in the input message of the operation that
904 ends the conversation
- 905 - The client or the service attempts to send a message in a conversation, after the
906 conversation has ended

907 This fault is named within the SCA namespace standard prefix "sca", which corresponds to URI
908 <http://www.oesa.org/xmlns/sca/1.0>.

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

913 Although conversational interfaces do not require that any identifying information be passed as
914 part of the body of messages, there is conceptually an identity associated with the conversation.
915 Individual implementations types MAY provide an API to access the ID associated with the
916 conversation, although no assumptions may be made about the structure of that identifier.
917 Implementation types MAY also provide a means to set the conversation ID by either the client or

918 the service provider, although the operation may only be supported by some binding/policy
919 combinations.

920

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

924

925 5.4 SCA-Specific Aspects for WSDL Interfaces

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

931 For WSDL interfaces, there is an extension mechanism that permits additional information to be
932 included within the WSDL document. SCA takes advantage of this extension mechanism. In order
933 to use the SCA extension mechanism, the SCA namespace (<http://www.oesa.org/xmlns/sca/1.0>)
934 must be declared within the WSDL document.

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

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

938

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

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

```
941 </simpleType>
```

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

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

```
948 requires="conversational"
```

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

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

954

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

957

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

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

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

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

```
962 </operation>
```

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

```
964 </operation>
```

965 ...
966 </portType>
967 ...
968

969

6 Composite

970
971
972
973

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.

974
975
976
977

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](#).

978
979
980
981
982

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](#).

983
984
985
986

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](#).

987

988
989

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.

990

991

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

992

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

993

```
<composite xmlns="http://www.oesa.org/xmlns/sca/1.0"
```

994

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

995

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

996

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

997

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

998

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

999

1000

```
<include name="xs:QName"/>*
```

1001

1002

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

1003

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

1004

```
<interface/>?
```

1005

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

1006

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

1007

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

1008

```
<callback>?
```

1009

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

1010

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

1011

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

1012

```
</callback>
```

```

1013     </service>
1014
1015     <reference name="xs:NCName" target="list of xs:anyURI"?
1016         promote="list of xs:anyURI" wiredByImpl="xs:boolean"?
1017         multiplicity="0..1 or 1..1 or 0..n or 1..n"?
1018         requires="list of xs:QName"? policySets="list of xs:QName"?>*
1019         <interface/>?
1020         <binding uri="xs:anyURI"? name="xs:QName"?
1021             requires="list of xs:QName"? policySets="list of
1022 xs:QName"?/>*
1023         <callback>?
1024             <binding uri="xs:anyURI"? name="xs:QName"?
1025                 requires="list of xs:QName"?
1026                 policySets="list of xs:QName"?/>+
1027         </callback>
1028     </reference>
1029
1030     <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
1031         many="xs:boolean"? mustSupply="xs:boolean"?>*
1032         default-property-value?
1033     </property>
1034
1035     <component name="xs:NCName" autowire="xs:boolean"?
1036         requires="list of xs:QName"? policySets="list of xs:QName"?>*
1037     <implementation/>?
1038     <service name="xs:NCName" requires="list of xs:QName"?
1039         policySets="list of xs:QName"?>*
1040     <interface/>?
1041     <binding uri="xs:anyURI"? name="xs:QName"?
1042         requires="list of xs:QName"?
1043         policySets="list of xs:QName"?/>*
1044     <callback>?
1045         <binding uri="xs:anyURI"? name="xs:QName"?
1046             requires="list of xs:QName"?
1047             policySets="list of xs:QName"?/>+
1048     </callback>
1049     </service>
1050     <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
1051         source="xs:string"? file="xs:anyURI"?>*
1052         property-value
1053     </property>
1054     <reference name="xs:NCName" target="list of xs:anyURI"?
1055         autowire="xs:boolean"? wiredByImpl="xs:boolean"?

```

```

1056         requires="list of xs:QName"? policySets="list of xs:QName"?
1057         multiplicity="0..1 or 1..1 or 0..n or 1..n"?/>*
1058     <interface/>?
1059     <binding uri="xs:anyURI"? name="xs:QName"?
1060         requires="list of xs:QName"?
1061         policySets="list of xs:QName"?/>*
1062     <callback>?
1063         <binding uri="xs:anyURI"? name="xs:QName"?
1064             requires="list of xs:QName"?
1065             policySets="list of xs:QName"?/>+
1066     </callback>
1067 </reference>
1068 </component>
1069
1070 <wire source="xs:anyURI" target="xs:anyURI" />*
1071
1072 </composite>
1073
1074
1075

```

The composite element has the following *attributes*:

- 1076 • **name (required)** – the name of the composite. The form of a composite name is an XML
1077 QName, in the namespace identified by the targetNamespace attribute.
- 1078 • **targetNamespace (optional)** – an identifier for a target namespace into which the
1079 composite is declared
- 1080 • **local (optional)** – whether all the components within the composite must all run in the
1081 same operating system process. local="true" means that all the components must run in
1082 the same process. local="false", which is the default, means that different components
1083 within the composite may run in different operating system processes and they may even
1084 run on different nodes on a network.
- 1085 • **autowire (optional)** – whether contained component references should be autowired, as
1086 described in [the Autowire section](#). Default is false.
- 1087 • **constrainingType (optional)** – the name of a constrainingType. When specified, the set
1088 of services, references and properties of the composite, plus related intents, is constrained
1089 to the set defined by the constrainingType. See [the ConstrainingType Section](#) for more
1090 details.
- 1091 • **requires (optional)** – a list of policy intents. See the [Policy Framework specification \[10\]](#)
1092 for a description of this attribute.
- 1093 • **policySets (optional)** – a list of policy sets. See the [Policy Framework specification \[10\]](#)
1094 for a description of this attribute.

1095 Composites contain **zero or more properties, services, components, references, wires and**
1096 **included composites**. These artifacts are described in detail in the following sections.

1097 Components contain configured implementations which hold the business logic of the composite.
1098 The components offer services and require references to other services. Composite services
1099 define the public services provided by the composite, which can be accessed from outside the
1100 composite. Composite references represent dependencies which the composite has on services
1101 provided elsewhere, outside the composite. Wires describe the connections between component
1102 services and component references within the composite. Included composites contribute the
1103 elements they contain to the using composite.

1104 Composite services involve the **promotion** of one service of one of the components within the
1105 composite, which means that the composite service is actually provided by one of the components
1106 within the composite. Composite references involve the **promotion** of one or more references of
1107 one or more components. Multiple component references can be promoted to the same composite
1108 reference, as long as all the component references are compatible with one another. Where
1109 multiple component references are promoted to the same composite reference, then they all share
1110 the same configuration, including the same target service(s).

1111 Composite services and composite references can use the configuration of their promoted services
1112 and references respectively (such as Bindings and Policy Sets). Alternatively composite services
1113 and composite references can override some or all of the configuration of the promoted services
1114 and references, through the configuration of bindings and other aspects of the composite service
1115 or reference.

1116 Component services and component references can be promoted to composite services and
1117 references and also be wired internally within the composite at the same time. For a reference,
1118 this only makes sense if the reference supports a multiplicity greater than 1.

1119 6.1 Property – Definition and Configuration

1120 **Properties** allow for the configuration of an implementation with externally set data values. An
1121 implementation, including a composite, can declare zero or more properties. Each property has a
1122 type, which may be either simple or complex. An implementation may also define a default value
1123 for a property. Properties are configured with values in the components that use the
1124 implementation.

1125 The declaration of a property in a composite follows the form described in the following schema
1126 snippet:

```
1127  
1128 <?xml version="1.0" encoding="ASCII"?>  
1129  
1130 <composite xmlns="http://www.oesa.org/xmlns/sca/1.0"  
1131 name="xs:QName" ... >  
1132  
1133 ...  
1134  
1135 <property name="xs:NCName" (type="xs:QName" | element="xs:QName")  
1136 many="xs:boolean"? mustSupply="xs:boolean"?>*  
1137 default-property-value?  
1138 </property>  
1139 ...  
1140  
1141 </composite>  
1142
```

1143 The property element has the following attributes:

- 1144 ▪ **name (required)** - the name of the property
- 1145 ▪ one of **(required)**:
 - 1146 ○ **type** – the type of the property - the qualified name of an XML schema type
 - 1147 ○ **element** – the type of the property defined as the qualified name of an XML
1148 schema global element – the type is the type of the global element

- 1149 ▪ **many (optional)** - whether the property is single-valued (false) or multi-valued (true).
1150 The default is **false**. In the case of a multi-valued property, it is presented to the
1151 implementation as a Collection of property values.
- 1152 ▪ **mustSupply (optional)** – whether the property value must be supplied by the
1153 component that uses the implementation – when mustSupply="true" the component must
1154 supply a value since the implementation has no default value for the property. A default-
1155 property-value should only be supplied when mustSupply="false" (the default setting for
1156 the mustSupply attribute), since the implication of a default value is that it is used only
1157 when a value is not supplied by the using component.

1158 The property element may contain an optional **default-property-value**, which provides default
1159 value for the property. The default value must match the type declared for the property:

- 1160 ○ a string, if **type** is a simple type (must match the **type** declared)
- 1161 ○ a complex type value matching the type declared by **type**
- 1162 ○ an element matching the element named by **element**
- 1163 ○ multiple values are permitted if many="true" is specified

1165 Implementation types other than **composite** can declare properties in an implementation-
1166 dependent form (eg annotations within a Java class), or through a property declaration of exactly
1167 the form described above in a componentType file.

1168 Property values can be configured when an implementation is used by a component. The form of
1169 the property configuration is shown in [the section on Components](#).

1170 6.1.1 Property Examples

1171
1172 For the following example of Property declaration and value setting, the following complex type is
1173 used as an example:

```
1174   <xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
1175              targetNamespace="http://foo.com/"
1176              xmlns:tns="http://foo.com/">
1177    <!-- ComplexProperty schema -->
1178    <xsd:element name="fooElement" type="MyComplexType"/>
1179    <xsd:complexType name="MyComplexType">
1180      <xsd:sequence>
1181        <xsd:element name="a" type="xsd:string"/>
1182        <xsd:element name="b" type="anyURI"/>
1183      </xsd:sequence>
1184      <attribute name="attr" type="xsd:string" use="optional"/>
1185    </xsd:complexType>
1186 </xsd:schema>
1187
```

1188 The following composite demonstrates the declaration of a property of a complex type, with a
1189 default value, plus it demonstrates the setting of a property value of a complex type within a
1190 component:

```
1191   <?xml version="1.0" encoding="ASCII"?>
1192
1193   <composite       xmlns="http://www.oesa.org/xmlns/sca/1.0"
1194                   xmlns:foo="http://foo.com">
```

```

1195         targetNamespace="http://foo.com"
1196         name="AccountServices">
1197 <!-- AccountServices Example1 -->
1198
1199     ...
1200
1201     <property name="complexFoo" type="foo:MyComplexType">
1202         <MyComplexPropertyValue xsi:type="foo:MyComplexType">
1203             <foo:a>AValue</foo:a>
1204             <foo:b>InterestingURI</foo:b>
1205         </MyComplexPropertyValue>
1206     </property>
1207
1208     <component name="AccountServiceComponent">
1209         <implementation.java class="foo.AccountServiceImpl"/>
1210         <property name="complexBar" source="$complexFoo"/>
1211         <reference name="accountDataService"
1212             target="AccountDataServiceComponent"/>
1213         <reference name="stockQuoteService" target="StockQuoteService"/>
1214     </component>
1215
1216     ...
1217
1218 </composite>

```

1219 In the declaration of the property named **complexFoo** in the composite **AccountServices**, the
1220 property is defined to be of type **foo:MyComplexType**. The namespace **foo** is declared in the
1221 composite and it references the example XSD, where **MyComplexType** is defined. The declaration
1222 of **complexFoo** contains a default value. This is declared as the content of the property element.
1223 In this example, the default value consists of the element **MyComplexPropertyValue** of type
1224 **foo:MyComplexType** and its two child elements **<foo:a>** and **<foo:b>**, following the definition of
1225 **MyComplexType**.

1226 In the component **AccountServiceComponent**, the component sets the value of the property
1227 **complexBar**, declared by the implementation configured by the component. In this case, the
1228 type of **complexBar** is **foo:MyComplexType**. The example shows that the value of the **complexBar**
1229 property is set from the value of the **complexFoo** property – the **source** attribute of the property
1230 element for **complexBar** declares that the value of the property is set from the value of a property
1231 of the containing composite. The value of the source attribute is **\$complexFoo**, where
1232 **complexFoo** is the name of a property of the composite. This value implies that the whole of the
1233 value of the source property is used to set the value of the component property.

1234 The following example illustrates the setting of the value of a property of a simple type (a string)
1235 from **part** of the value of a property of the containing composite which has a complex type:

```

1236 <?xml version="1.0" encoding="ASCII"?>
1237
1238 <composite xmlns="http://www.oesa.org/xmlns/sca/1.0"
1239     xmlns:foo="http://foo.com"
1240     targetNamespace="http://foo.com"
1241     name="AccountServices">

```

```

1242 <!-- AccountServices Example2 -->
1243
1244 ...
1245
1246 <property name="complexFoo" type="foo:MyComplexType">
1247     <MyComplexPropertyValue xsi:type="foo:MyComplexType">
1248         <foo:a>AValue</foo:a>
1249         <foo:b>InterestingURI</foo:b>
1250     </MyComplexPropertyValue>
1251 </property>
1252
1253 <component name="AccountServiceComponent">
1254     <implementation.java class="foo.AccountServiceImpl"/>
1255     <property name="currency" source="$complexFoo/a"/>
1256     <reference name="accountDataService"
1257         target="AccountDataServiceComponent"/>
1258     <reference name="stockQuoteService" target="StockQuoteService"/>
1259 </component>
1260
1261 ...
1262
1263 </composite>

```

1264 In this example, the component **AccountServiceComponent** sets the value of a property called
1265 **currency**, which is of type string. The value is set from a property of the composite
1266 **AccountServices** using the source attribute set to **\$complexFoo/a**. This is an XPath expression
1267 that selects the property name **complexFoo** and then selects the value of the **a** subelement of
1268 complexFoo. The "a" subelement is a string, matching the type of the currency property.

1269 Further examples of declaring properties and setting property values in a component follow:

1270 Declaration of a property with a simple type and a default value:

```

1271 <property name="SimpleTypeProperty" type="xsd:string">
1272 MyValue
1273 </property>

```

1274

1275 Declaration of a property with a complex type and a default value:

```

1276 <property name="complexFoo" type="foo:MyComplexType">
1277     <MyComplexPropertyValue xsi:type="foo:MyComplexType">
1278         <foo:a>AValue</foo:a>
1279         <foo:b>InterestingURI</foo:b>
1280     </MyComplexPropertyValue>
1281 </property>

```

1282

1283 Declaration of a property with an element type:

```

1284 <property name="elementFoo" element="foo:fooElement">
1285 <foo:fooElement>

```

1286 <foo:a>AValue</foo:a>
1287 <foo:b>InterestingURI</foo:b>
1288 </foo:fooElement>
1289 </property>

1290

1291 Property value for a simple type:

1292 <property name="SimpleTypeProperty">
1293 MyValue
1294 </property>

1295

1296

1297 Property value for a complex type, also showing the setting of an attribute value of the complex
1298 type:

1299 <property name="complexFoo">
1300 <MyComplexPropertyValue xsi:type="foo:MyComplexType" attr="bar">
1301 <foo:a>AValue</foo:a>
1302 <foo:b>InterestingURI</foo:b>
1303 </MyComplexPropertyValue>
1304 </property>

1305

1306 Property value for an element type:

1307 <property name="elementFoo">
1308 <foo:fooElement attr="bar">
1309 <foo:a>AValue</foo:a>
1310 <foo:b>InterestingURI</foo:b>
1311 </foo:fooElement>
1312 </property>

1313

1314 Declaration of a property with a complex type where multiple values are supported:

1315 <property name="complexFoo" type="foo:MyComplexType" many="true"/>
1316

1317 Setting of a value for that property where multiple values are supplied:

1318 <property name="complexFoo">
1319 <MyComplexPropertyValue1 xsi:type="foo:MyComplexType" attr="bar">
1320 <foo:a>AValue</foo:a>
1321 <foo:b>InterestingURI</foo:b>
1322 </MyComplexPropertyValue1>
1323 <MyComplexPropertyValue2 xsi:type="foo:MyComplexType" attr="zing">
1324 <foo:a>BValue</foo:a>
1325 <foo:b>BoringURI</foo:b>
1326 </MyComplexPropertyValue2>
1327 </property>
1328

1329

1330 6.2 References

1331 The *references of a composite* are defined by *promoting* references defined by components
1332 contained in the composite. Each promoted reference indicates that the component reference
1333 must be resolved by services outside the composite. A component reference is promoted using a
1334 composite *reference element*.

1335 A composite reference is represented by a *reference element* which is a child of a composite
1336 element. There can be *zero or more* *reference* elements in a composite. The following snippet
1337 shows the composite schema with the schema for a *reference* element.

1338

```
1339 <?xml version="1.0" encoding="ASCII"?>
1340 <!-- Reference schema snippet -->
1341 <composite xmlns="http://www.osea.org/xmlns/sca/1.0"
1342           targetNamespace="xs:anyURI"
1343           name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?
1344           constrainingType="QName"?
1345           requires="list of xs:QName"? policySets="list of
1346 xs:QName"?>
1347
1348     ...
1349
1350     <reference name="xs:NCName" target="list of xs:anyURI"?
1351               promote="list of xs:anyURI" wiredByImpl="xs:boolean"?
1352               multiplicity="0..1 or 1..1 or 0..n or 1..n"?
1353               requires="list of xs:QName"? policySets="list of xs:QName"?>*
1354     <interface/>?
1355     <binding uri="xs:anyURI"? name="xs:QName"?
1356             requires="list of xs:QName" policySets="list of
1357 xs:QName"?/>*
1358     <callback?>
1359       <binding uri="xs:anyURI"? name="xs:QName"?
1360             requires="list of xs:QName"?
1361             policySets="list of xs:QName"?/>+
1362     </callback>
1363 </reference>
1364
1365     ...
1366
1367 </composite>
```

1370 The *reference* element has the following *attributes*:

- 1371 • **name (required)** – the name of the reference. The name must be unique across all the
1372 composite references in the composite. The name of the composite reference can be
1373 different then the name of the promoted component reference.
- 1374 • **promote (required)** – identifies one or more promoted component references. The value
1375 is a list of values of the form <component-name>/<reference-name> separated by
1376 spaces. The specification of the reference name is optional if the component has only one
1377 reference.
- 1378 • **requires (optional)** – a list of required policy intents. See the [Policy Framework](#)
1379 [specification \[10\]](#) for a description of this attribute.
- 1380 • **policySets (optional)** – a list of policy sets. See the [Policy Framework specification \[10\]](#)
1381 for a description of this attribute.
- 1382 • **multiplicity (optional)** - Defines the number of wires that can connect the reference to
1383 target services. The multiplicity can have the following values
 - 1384 ○ 1..1 – one wire can have the reference as a source
 - 1385 ○ 0..1 – zero or one wire can have the reference as a source
 - 1386 ○ 1..n – one or more wires can have the reference as a source
 - 1387 ○ 0..n - zero or more wires can have the reference as a source
- 1388 • **target (optional)** – a list of one or more of target service URI's, depending on multiplicity
1389 setting. Each value wires the reference to a service in a composite that uses the composite
1390 containing the reference as an implementation for one of its components. For more details
1391 on wiring see [the section on Wires](#).
- 1392 • **wiredByImpl (optional)** – a boolean value, "false" by default, which indicates that the
1393 implementation wires this reference dynamically. If set to "true" it indicates that the
1394 target of the reference is set at runtime by the implementation code (eg by the code
1395 obtaining an endpoint reference by some means and setting this as the target of the
1396 reference through the use of programming interfaces defined by the relevant Client and
1397 Implementation specification). If "true" is set, then the reference should not be wired
1398 statically within a using composite, but left unwired.

1399
1400 The composite reference can optionally specify an **interface**, **multiplicity**, **required intents** ,
1401 and **bindings**. Whatever is not specified is defaulted from the promoted component reference(s).

1402
1403 If an **interface** is specified it must provide an interface which is the same or which is a compatible
1404 superset of the interface declared by the promoted component reference, i.e. provide a superset
1405 of the operations defined by the component for the reference. The interface is described by **zero**
1406 **or one interface element** which is a child element of the reference element. For details on the
1407 interface element see [the Interface section](#).

1408
1409 The value specified for the **multiplicity** attribute has to be compatible with the multiplicity
1410 specified on the component reference, i.e. it has to be equal or further restrict. So a composite
1411 reference of multiplicity 0..1 or 1..1 can be used where the promoted component reference has
1412 multiplicity 0..n and 1..n respectively. However, a composite reference of multiplicity 0..n or 1..n
1413 cannot be used to promote a component reference of multiplicity 0..1 or 1..1 respectively.

1414
1415 Specified **required intents** add to or further qualify the required intents defined for the promoted
1416 component reference.

1417
1418 If one or more **bindings** are specified they **override** any and all of the bindings defined for the
1419 promoted component reference from the composite reference perspective. The bindings defined on

1420 the component reference are still in effect for local wires within the composite that have the
1421 component reference as their source. A reference element has zero or more **binding elements** as
1422 children. Details of the binding element are described in the [Bindings section](#). For more details on
1423 wiring see [the section on Wires](#).

1424 Note that a binding element may specify an endpoint which is the target of that binding. A
1425 reference must not mix the use of endpoints specified via binding elements with target endpoints
1426 specified via the target attribute. If the target attribute is set, then binding elements can only list
1427 one or more binding types that can be used for the wires identified by the target attribute. All the
1428 binding types identified are available for use on each wire in this case. If endpoints are specified
1429 in the binding elements, each endpoint must use the binding type of the binding element in which
1430 it is defined. In addition, each binding element needs to specify an endpoint in this case.

1431 A **reference** element has an optional **callback** element used if the interface has a callback
1432 defined, which has one or more **binding** elements as children. The **callback** and its binding child
1433 elements are specified if there is a need to have binding details used to handle callbacks. If the
1434 callback element is not present, the behaviour is runtime implementation dependent.

1435

1436 The same component reference maybe promoted more than once, using different composite
1437 references, but only if the multiplicity defined on the component reference is 0..n or 1..n. The
1438 multiplicity on the composite reference can restrict accordingly.

1439 Two or more component references may be promoted by one composite reference, but only when

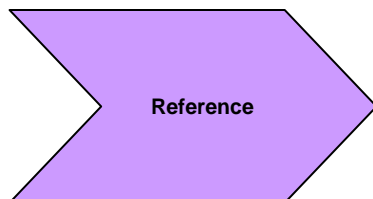
- 1440 • the interfaces of the component references are the same, or if the composite reference
1441 itself declares an interface then all the component references must have interfaces which
1442 are compatible with the composite reference interface
- 1443 • the multiplicities of the component references are compatible, i.e one can be the restricted
1444 form of the another, which also means that the composite reference carries the restricted
1445 form either implicitly or explicitly
- 1446 • the intents declared on the component references must be compatible – the intents which
1447 apply to the composite reference in this case are the union of the required intents
1448 specified for each of the promoted component references. If any intents contradict (eg
1449 mutually incompatible qualifiers for a particular intent) then there is an error.

1450

1451 6.2.1 Example Reference

1452

1453 The following figure shows the reference symbol that is used to represent a reference in an
1454 assembly diagram.

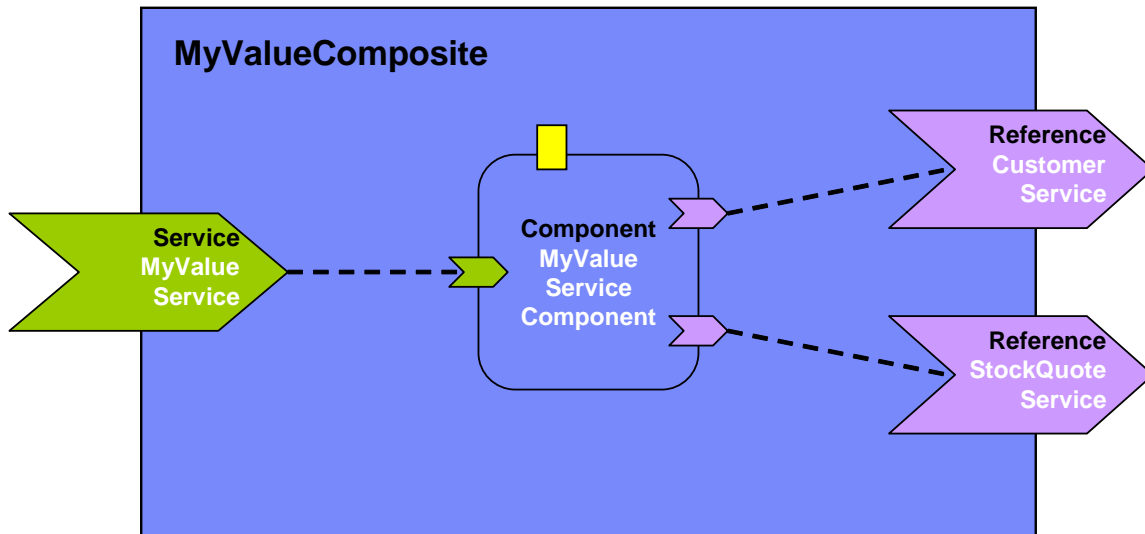


1455

1456 *Figure 7: Reference symbol*

1457 The following figure shows the assembly diagram for the MyValueComposite containing the
1458 reference CustomerService and the reference StockQuoteService.

1459



1460
1461 *Figure 8: MyValueComposite showing References*
1462

1463 The following snippet shows the MyValueComposite.composite file for the MyValueComposite
1464 containing the reference elements for the CustomerService and the StockQuoteService. The
1465 reference CustomerService is bound using the SCA binding. The reference StockQuoteService is
1466 bound using the Web service binding. The endpoint addresses of the bindings can be specified, for
1467 example using the binding *uri* attribute (for details see the [Bindings](#) section), or overridden in an
1468 enclosing composite. Although in this case the reference StockQuoteService is bound to a Web
1469 service, its interface is defined by a Java interface, which was created from the WSDL portType of
1470 the target web service.

```
1471
1472 <?xml version="1.0" encoding="ASCII"?>
1473 <!-- MyValueComposite_3 example -->
1474 <composite xmlns="http://www.oesa.org/xmlns/sca/1.0"
1475           targetNamespace="http://foo.com"
1476           name="MyValueComposite" >
1477
1478   ...
1479
1480   <component name="MyValueServiceComponent">
1481     <implementation.java
1482     class="services.myvalue.MyValueServiceImpl"/>
1483     <property name="currency">EURO</property>
1484     <reference name="customerService"/>
1485     <reference name="StockQuoteService"/>
1486   </component>
1487
1488   <reference name="CustomerService"
1489             promote="MyValueServiceComponent/customerService">
1490     <interface.java interface="services.customer.CustomerService"/>
1491     <!-- The following forces the binding to be binding.sca whatever
1492     is -->
```

```

1493         <!-- specified by the component reference or by the underlying
1494 -->
1495         <!-- implementation
1496 -->
1497         <binding.sca/>
1498     </reference>
1499
1500     <reference name="StockQuoteService"
1501         promote="MyValueServiceComponent/StockQuoteService">
1502         <interface.java
1503 interface="services.stockquote.StockQuoteService"/>
1504         <binding.ws port="http://www.stockquote.org/StockQuoteService#
1505 wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
1506     </reference>
1507
1508     ...
1509
1510
1511 </composite>
1512

```

1513 6.3 Service

1514 The *services of a composite* are defined by promoting services defined by components
1515 contained in the composite. A component service is promoted by means of a composite *service*
1516 *element*.

1517 A composite service is represented by a *service element* which is a child of the composite
1518 element. There can be *zero or more* service elements in a composite. The following snippet
1519 shows the composite schema with the schema for a service child element:

```

1520
1521 <?xml version="1.0" encoding="ASCII"?>
1522 <!-- Servicee schema snippet -->
1523 <composite xmlns="http://www.oesa.org/xmlns/sca/1.0"
1524     targetNamespace="xs:anyURI"
1525     name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?
1526     constrainingType="QName"?
1527     requires="list of xs:QName"? policySets="list of
1528 xs:QName"?>
1529     ...
1530
1531     <service name="xs:NCName" promote="xs:anyURI"
1532         requires="list of xs:QName"? policySets="list of xs:QName"?>*
1533     <interface/>?
1534     <binding uri="xs:anyURI"? name="xs:QName"?
1535         requires="list of xs:QName" policySets="list of
1536 xs:QName"?/>*
1537     <callback?>

```

```

1538         <binding uri="xs:anyURI"? name="xs:QName"?
1539             requires="list of xs:QName"?
1540             policySets="list of xs:QName"?/>+
1541     </callback>
1542 </service>
1543
1544     ...
1545
1546 </composite>
1547

```

1548 The service element has the following *attributes*:

- 1549 • **name (required)** – the name of the service, the name MUST BE unique across all the
1550 composite services in the composite. The name of the composite service can be different
1551 from the name of the promoted component service.
- 1552 • **promote (required)** – identifies the promoted service, the value is of the form
1553 <component-name>/<service-name>. The service name is optional if the target
1554 component only has one service.
- 1555 • **requires (optional)** – a list of required policy intents. See the [Policy Framework](#)
1556 [specification \[10\]](#) for a description of this attribute.
- 1557 • **policySets (optional)** – a list of policy sets. See the [Policy Framework specification \[10\]](#)
1558 for a description of this attribute.

1559

1560 The composite service can optionally specify an *interface*, *required intents* and *bindings*.
1561 Whatever is not specified is defaulted from the promoted component service.

1562

1563 If an *interface* is specified it must be the same or a compatible subset of the interface provided
1564 by the promoted component service, i.e. provide a subset of the operations defined by the
1565 component service. The interface is described by **zero or one interface element** which is a child
1566 element of the service element. For details on the interface element see [the Interface section](#).

1567

1568 Specified **required intents** add to or further qualify the required intents defined by the promoted
1569 component service.

1570

1571 If bindings are specified they **override** the bindings defined for the promoted component service
1572 from the composite service perspective. The bindings defined on the component service are still in
1573 effect for local wires within the composite that target the component service. A service element
1574 has zero or more **binding elements** as children. Details of the binding element are described in
1575 the [Bindings section](#). For more details on wiring see [the Wiring section](#).

1576 A service element has an optional **callback** element used if the interface has a callback defined,,
1577 which has one or more **binding** elements as children. The **callback** and its binding child
1578 elements are specified if there is a need to have binding details used to handle callbacks. If the
1579 callback element is not present, the behaviour is runtime implementation dependent.

1580

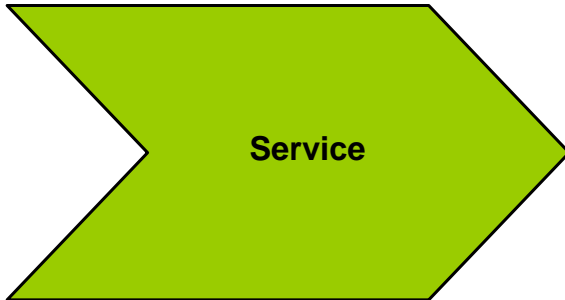
1581 The same component service can be promoted by more than one composite service.

1582

1583 **6.3.1 Service Examples**

1584

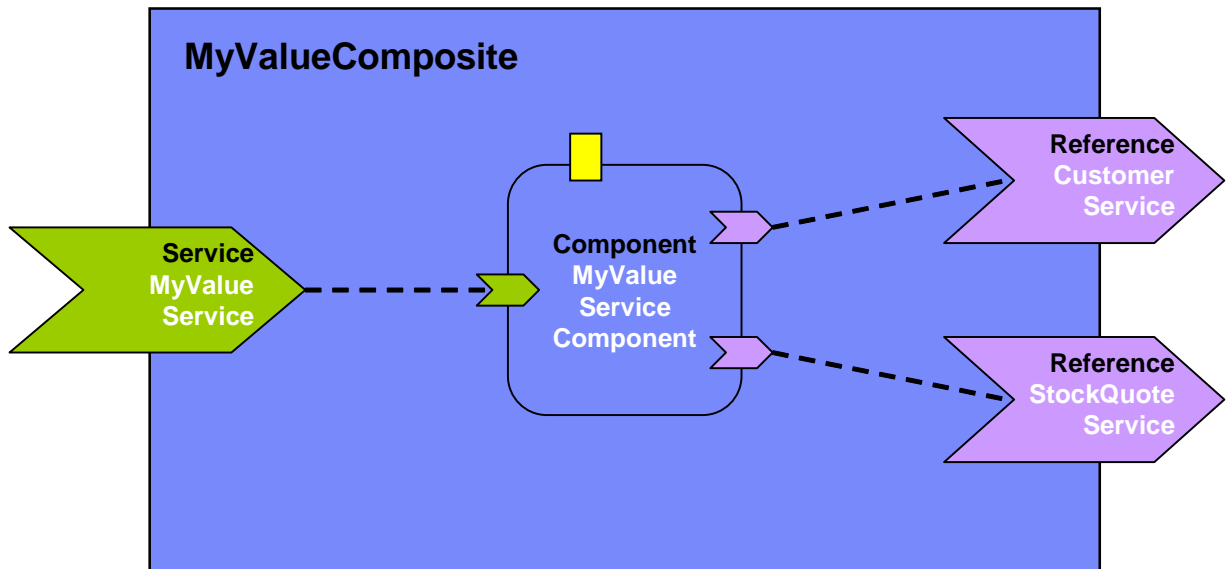
1585 The following figure shows the service symbol that used to represent a service in an assembly
1586 diagram:



1587

1588 *Figure 9: Service symbol*

1589 The following figure shows the assembly diagram for the MyValueComposite containing the service
1590 MyValueService.



1591

1592 *Figure 10: MyValueComposite showing Service*

1593

1594 The following snippet shows the MyValueComposite.composite file for the MyValueComposite
1595 containing the service element for the MyValueService, which is a promote of the service offered
1596 by the MyValueServiceComponent. The name of the promoted service is omitted since
1597 MyValueServiceComponent offers only one service. The composite service MyValueService is
1598 bound using a Web service binding.

1599

```
1600 <?xml version="1.0" encoding="ASCII"?>  
1601 <!-- MyValueComposite_4 example -->  
1602 <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"  
1603 targetNamespace="http://foo.com"  
1604 name="MyValueComposite" >
```

1605

```

1606     ...
1607
1608     <service name="MyValueService" promote="MyValueServiceComponent">
1609         <interface.java interface="services.myvalue.MyValueService"/>
1610         <binding.ws port="http://www.myvalue.org/MyValueService#
1611             wsdl.endpoint(MyValueService/MyValueServiceSOAP)"/>
1612     </service>
1613
1614     <component name="MyValueServiceComponent">
1615         <implementation.java
1616 class="services.myvalue.MyValueServiceImpl"/>
1617         <property name="currency">EURO</property>
1618         <service name="MyValueService"/>
1619         <reference name="customerService"/>
1620         <reference name="StockQuoteService"/>
1621     </component>
1622
1623     ...
1624
1625 </composite>
1626

```

1627 6.4 Wire

1628 **SCA wires** within a composite connect *source component references* to *target component*
1629 *services*.

1630 One way of defining a wire is by **configuring a reference of a component using its target**
1631 **attribute**. The reference element is configured with the wire-target-URI of the service(s) that
1632 resolve the reference. Multiple target services are valid when the reference has a multiplicity of
1633 0..n or 1..n.

1634 An alternative way of defining a Wire is by means of a **wire element** which is a child of the
1635 composite element. There can be **zero or more** wire elements in a composite. This alternative
1636 method for defining wires is useful in circumstances where separation of the wiring from the
1637 elements the wires connect helps simplify development or operational activities. An example is
1638 where the components used to build a domain are relatively static but where new or changed
1639 applications are created regularly from those components, through the creation of new assemblies
1640 with different wiring. Deploying the wiring separately from the components allows the wiring to
1641 be created or modified with minimum effort.

1642 Note that a Wire specified via a wire element is equivalent to a wire specified via the target
1643 attribute of a reference. The rule which forbids mixing of wires specified with the target attribute
1644 with the specification of endpoints in binding subelements of the reference also applies to wires
1645 specified via separate wire elements.

1646 The following snippet shows the composite schema with the schema for the reference elements of
1647 components and composite services and the wire child element:

```

1648
1649 <?xml version="1.0" encoding="ASCII"?>
1650 <!-- Wires schema snippet -->
1651 <composite xmlns="http://www.oesa.org/xmlns/sca/1.0"

```



```

1652         targetNamespace="xs:anyURI"
1653         name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?
1654         constrainingType="QName"?
1655         requires="list of xs:QName"? policySets="list of
1656 xs:QName"?>
1657
1658     ...
1659
1660     <wire source="xs:anyURI" target="xs:anyURI" />*
1661
1662 </composite>

```

1665 The *reference element of a component* and the *reference element of a service* has a list of
1666 one or more of the following *wire-target-URI* values for the target, with multiple values
1667 separated by a space:

- 1668 • `<component-name>/<service-name>`
 - 1669 ○ where the target is a service of a component. The specification of the service
1670 name is optional if the target component only has one service with a compatible
1671 interface

1672

1673 The *wire element* has the following attributes:

- 1674 • **source (required)** – names the source component reference. Valid URI schemes are:
 - 1675 ○ `<component-name>/<reference-name>`
 - 1676 ▪ where the source is a component reference. The specification of the
1677 reference name is optional if the source component only has one reference
- 1678 • **target (required)** – names the target component service. Valid URI schemes are
 - 1679 ○ `<component-name>/<service-name>`
 - 1680 ▪ where the target is a service of a component. The specification of the
1681 service name is optional if the target component only has one service with
1682 a compatible interface

1683 For a composite used as a component implementation, wires can only link sources and targets
1684 that are contained in the same composite (irrespective of which file or files are used to describe
1685 the composite). Wiring to entities outside the composite is done through services and references
1686 of the composite with wiring defined by the next higher composite.

1687 A wire may only connect a source to a target if the target implements an interface that is
1688 compatible with the interface required by the source. The source and the target are compatible if:

- 1689 1. the source interface and the target interface MUST either both be remotable or they are
1690 both local
- 1691 2. the operations on the target interface MUST be the same as or be a superset of the
1692 operations in the interface specified on the source
- 1693 3. compatibility for the individual operation is defined as compatibility of the signature, that
1694 is operation name, input types, and output types MUST BE the same.
- 1695 4. the order of the input and output types also MUST BE the same.
- 1696 5. the set of Faults and Exceptions expected by the source MUST BE the same or be a
1697 superset of those specified by the target.

1698 6. other specified attributes of the two interfaces MUST match, including Scope and Callback
1699 interface

1700 A Wire can connect between different interface languages (eg. Java interfaces and WSDL
1701 portTypes) in either direction, as long as the operations defined by the two interface types are
1702 equivalent. They are equivalent if the operation(s), parameter(s), return value(s) and
1703 faults/exceptions map to each other.

1704 Service clients cannot (portably) ask questions at runtime about additional interfaces that are
1705 provided by the implementation of the service (e.g. the result of "instance of" in Java is non
1706 portable). It is valid for an SCA implementation to have proxies for all wires, so that, for example,
1707 a reference object passed to an implementation may only have the business interface of the
1708 reference and may not be an instance of the (Java) class which is used to implement the target
1709 service, even where the interface is local and the target service is running in the same process.

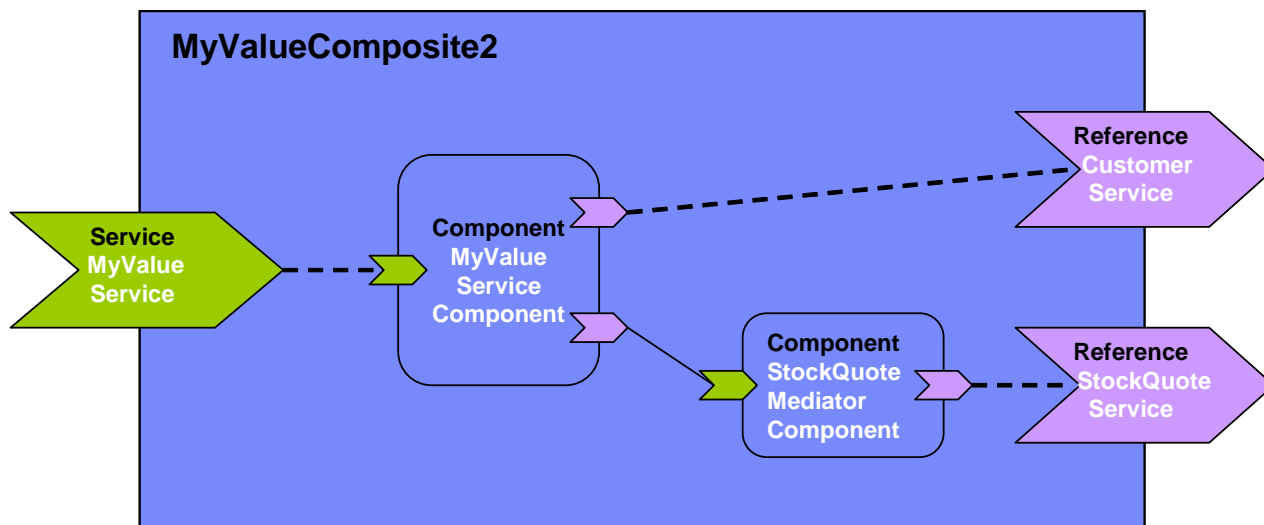
1710 **Note:** It is permitted to deploy a composite that has references that are not wired. For the case of
1711 an un-wired reference with multiplicity 1..1 or 1..n the deployment process provided by an SCA
1712 runtime SHOULD issue a warning.

1713

1714 6.4.1 Wire Examples

1715

1716 The following figure shows the assembly diagram for the MyValueComposite2 containing wires
1717 between service, components and references.



1718

1719 *Figure 11: MyValueComposite2 showing Wires*

1720

1721 The following snippet shows the MyValueComposite2.composite file for the MyValueComposite2
1722 containing the configured component and service references. The service MyValueService is wired
1723 to the MyValueServiceComponent. The MyValueServiceComponent's customerService reference is
1724 wired to the composite's CustomerService reference. The MyValueServiceComponent's
1725 stockQuoteService reference is wired to the StockQuoteMediatorComponent, which in turn has its
1726 reference wired to the StockQuoteService reference of the composite.

1727

```
1728 <?xml version="1.0" encoding="ASCII"?>  
1729 <!-- MyValueComposite Wires examples -->  
1730 <composite xmlns="http://www.oesa.org/xmlns/sca/1.0"  
1731 targetNamespace="http://foo.com"
```

```

1732         name="MyValueComposite2" >
1733
1734     <service name="MyValueService" promote="MyValueServiceComponent">
1735         <interface.java interface="services.myvalue.MyValueService"/>
1736         <binding.ws port="http://www.myvalue.org/MyValueService#
1737             wsdl.endpoint(MyValueService/MyValueServiceSOAP)"/>
1738     </service>
1739
1740     <component name="MyValueServiceComponent">
1741         <implementation.java
1742 class="services.myvalue.MyValueServiceImpl"/>
1743         <property name="currency">EURO</property>
1744         <service name="MyValueService"/>
1745         <reference name="customerService"/>
1746         <reference name="stockQuoteService"
1747             target="StockQuoteMediatorComponent"/>
1748     </component>
1749
1750     <component name="StockQuoteMediatorComponent">
1751         <implementation.java class="services.myvalue.SQMediatorImpl"/>
1752         <property name="currency">EURO</property>
1753         <reference name="stockQuoteService"/>
1754     </component>
1755
1756     <reference name="CustomerService"
1757         promote="MyValueServiceComponent/customerService">
1758         <interface.java interface="services.customer.CustomerService"/>
1759         <binding.sca/>
1760     </reference>
1761
1762     <reference name="StockQuoteService"
1763 promote="StockQuoteMediatorComponent">
1764         <interface.java
1765 interface="services.stockquote.StockQuoteService"/>
1766         <binding.ws port="http://www.stockquote.org/StockQuoteService#
1767             wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
1768     </reference>
1769
1770 </composite>
1771

```

1772 6.4.2 Autowire

1773 SCA provides a feature named **Autowire**, which can help to simplify the assembly of composites.
1774 Autowire enables component references to be automatically wired to component services which
1775 will satisfy those references, without the need to create explicit wires between the references and

1776 the services. When the autowire feature is used, a component reference which is not promoted
1777 and which is not explicitly wired to a service within a composite is automatically wired to a target
1778 service within the same composite. Autowire works by searching within the composite for a
1779 service interface which matches the interface of the references.

1780 The autowire feature is not used by default. Autowire is enabled by the setting of an autowire
1781 attribute to "true". Autowire is disabled by setting of the autowire attribute to "false" The autowire
1782 attribute can be applied to any of the following elements within a composite:

- 1783 • reference
- 1784 • component
- 1785 • composite

1786 Where an element does not have an explicit setting for the autowire attribute, it inherits the
1787 setting from its parent element. Thus a reference element inherits the setting from its containing
1788 component. A component element inherits the setting from its containing composite. Where
1789 there is no setting on any level, autowire="false" is the default.

1790 As an example, if a composite element has autowire="true" set, this means that autowiring is
1791 enabled for all component references within that composite. In this example, autowiring can be
1792 turned off for specific components and specific references through setting autowire="false" on the
1793 components and references concerned.

1794 For each component reference for which autowire is enabled, the autowire process searches within
1795 the composite for target services which are compatible with the reference. "Compatible" here
1796 means:

- 1797 • the target service interface must be a compatible superset of the reference interface (as
1798 defined in [the section on Wires](#))
- 1799 • the intents, bindings and policies applied to the service must be compatible on the
1800 reference – so that wiring the reference to the service will not cause an error due to
1801 binding and policy mismatch (see [the Policy Framework specification \[10\]](#) for details)

1802 If the search finds *more than 1* valid target service for a particular reference, the action taken
1803 depends on the multiplicity of the reference:

- 1804 • for multiplicity 0..1 and 1..1, the SCA runtime selects one of the target services in a
1805 runtime-dependent fashion and wires the reference to that target service
- 1806 • for multiplicity 0..n and 1..n, the reference is wired to all of the target services

1807 If the search finds *no* valid target services for a particular reference, the action taken depends on
1808 the multiplicity of the reference:

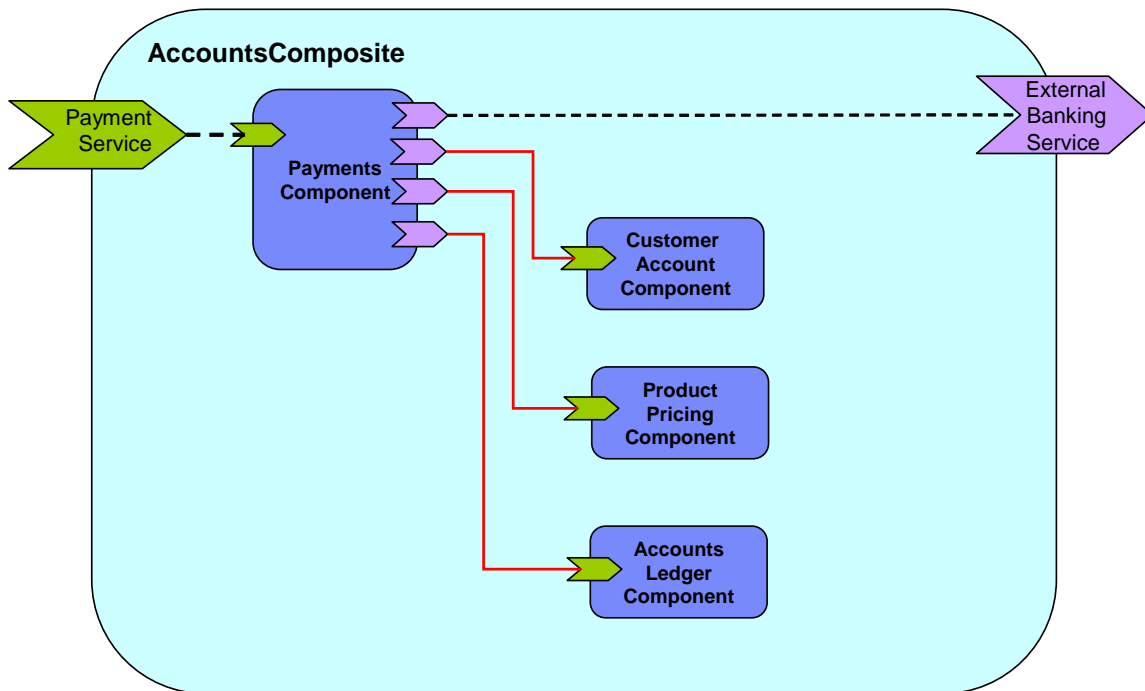
- 1809 • for multiplicity 0..1 and 0..n, there is no problem – no services are wired and there is no
1810 error
- 1811 • for multiplicity 1..1 and 1..n, an error is raised by the SCA runtime since the reference is
1812 intended to be wired

1813

1814 6.4.3 Autowire Examples

1815 This example demonstrates two versions of the same composite – the first version is done using
1816 explicit wires, with no autowiring used, the second version is done using autowire. In both cases
1817 the end result is the same – the same wires connect the references to the services.

1818 First, here is a diagram for the composite:



1819

1820 *Figure 12: Example Composite for Autowire*

1821 First, the composite using explicit wires:

```

1822 <?xml version="1.0" encoding="UTF-8"?>
1823 <!-- Autowire Example - No autowire -->
1824 <composite xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
1825           xmlns="http://www.oesa.org/xmlns/sca/1.0"
1826           targetNamespace="http://foo.com"
1827           name="AccountComposite">
1828
1829   <service name="PaymentService" promote="PaymentsComponent" />
1830
1831   <component name="PaymentsComponent">
1832     <implementation.java class="com.foo.accounts.Payments" />
1833     <service name="PaymentService" />
1834     <reference name="CustomerAccountService"
1835               target="CustomerAccountComponent" />
1836     <reference name="ProductPricingService"
1837               target="ProductPricingComponent" />
1838     <reference name="AccountsLedgerService"
1839               target="AccountsLedgerComponent" />
1840     <reference name="ExternalBankingService" />
1841   </component>
1842
1843   <component name="CustomerAccountComponent">
1844     <implementation.java class="com.foo.accounts.CustomerAccount" />

```

```

1845     </component>
1846
1847     <component name="ProductPricingComponent">
1848         <implementation.composite class="com.foo.accounts.ProductPricing"/>
1849     </component>
1850
1851     <component name="AccountsLedgerComponent">
1852         <implementation.composite class="com.foo.accounts.AccountsLedger"/>
1853     </component>
1854
1855     <reference name="ExternalBankingService"
1856         promote="PaymentsComponent/ExternalBankingService"/>
1857
1858 </composite>
1859
1860 Secondly, the composite using autowire:
1861 <?xml version="1.0" encoding="UTF-8"?>
1862 <!-- Autowire Example - With autowire -->
1863 <composite xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
1864     xmlns="http://www.oesa.org/xmlns/sca/1.0"
1865     targetNamespace="http://foo.com"
1866     name="AccountComposite">
1867
1868     <service name="PaymentService" promote="PaymentsComponent">
1869         <interface.java class="com.foo.PaymentServiceInterface"/>
1870     </service>
1871
1872     <component name="PaymentsComponent" autowire="true">
1873         <implementation.java class="com.foo.accounts.Payments"/>
1874         <service name="PaymentService"/>
1875         <reference name="CustomerAccountService"/>
1876         <reference name="ProductPricingService"/>
1877         <reference name="AccountsLedgerService"/>
1878         <reference name="ExternalBankingService"/>
1879     </component>
1880
1881     <component name="CustomerAccountComponent">
1882         <implementation.java class="com.foo.accounts.CustomerAccount"/>
1883     </component>
1884
1885     <component name="ProductPricingComponent">
1886         <implementation.composite
1887     class="com.foo.accounts.ProductPricing"/>

```

```

1888     </component>
1889
1890     <component name="AccountsLedgerComponent">
1891         <implementation.composite
1892 class="com.foo.accounts.AccountsLedger"/>
1893     </component>
1894
1895     <reference name="ExternalBankingService"
1896         promote="PaymentsComponent/ExternalBankingService"/>
1897
1898 </composite>
1899 In this second case, autowire is set on for the PaymentsComponent and there are no explicit wires
1900 for any of its references – the wires are created automatically through autowire.
1901 Note: In the second example, it would be possible to omit all of the service and reference
1902 elements from the PaymentsComponent. They are left in for clarity, but if they are omitted, the
1903 component service and references still exist, since they are provided by the implementation used
1904 by the component.

```

1905

1906 6.5 Using Composites as Component Implementations

1907 Composites may form *component implementations* in higher-level composites – in other words
1908 the higher-level composites can have components which are implemented by composites.

1909 When a composite is used as a component implementation, it defines a boundary of visibility.
1910 Components within the composite cannot be referenced directly by the using component. The
1911 using component can only connect wires to the services and references of the used composite and
1912 set values for any properties of the composite. The internal construction of the composite is
1913 invisible to the using component.

1914 A composite used as a component implementation must also honor a *completeness contract*.
1915 The services, references and properties of the composite form a contract which is relied upon by
1916 the using component. The concept of completeness of the composite implies:

- 1917 • the composite must have at least one service or at least one reference.
1918 A component with no services and no references is not meaningful in terms of SCA, since
1919 it cannot be wired to anything – it neither provides nor consumes any services
1920
- 1921 • each service offered by the composite must be wired to a service of a component or to a
1922 composite reference.
1923 If services are left unwired, the implication is that some exception will occur at runtime if
1924 the service is invoked.

1925 The component type of a composite is defined by the set of service elements, reference elements
1926 and property elements that are the children of the composite element.

1927 Composites are used as component implementations through the use of the
1928 *implementation.composite* element as a child element of the component. The schema snippet
1929 for the implementation.composite element is:

1930

```

1931 <?xml version="1.0" encoding="ASCII"?>
1932 <!-- Composite Implementation schema snippet -->
1933 <composite xmlns="http://www.oesa.org/xmlns/sca/1.0"
1934     targetNamespace="xs:anyURI"

```

```

1935         name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?
1936         constrainingType="QName"?
1937         requires="list of xs:QName"? policySets="list of
1938 xs:QName"?>
1939
1940     ...
1941
1942     <component name="xs:NCName" autowire="xs:boolean"?
1943         requires="list of xs:QName"? policySets="list of xs:QName"?>*
1944         <implementation.composite name="xs:QName"/>?
1945         <service name="xs:NCName" requires="list of xs:QName"?
1946             policySets="list of xs:QName"?>*
1947             <interface/>?
1948             <binding uri="xs:anyURI" name="xs:QName"?
1949                 requires="list of xs:QName"
1950                 policySets="list of xs:QName"?/>*
1951             <callback>?
1952                 <binding uri="xs:anyURI"? name="xs:QName"?
1953                     requires="list of xs:QName"?
1954                     policySets="list of xs:QName"?/>+
1955             </callback>
1956         </service>
1957         <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
1958             source="xs:string"? file="xs:anyURI"?>*
1959             property-value
1960         </property>
1961         <reference name="xs:NCName" target="list of xs:anyURI"?
1962             autowire="xs:boolean"? wiredByImpl="xs:boolean"?
1963             requires="list of xs:QName"? policySets="list of xs:QName"?
1964             multiplicity="0..1 or 1..1 or 0..n or 1..n"?/>*
1965         <interface/>?
1966         <binding uri="xs:anyURI"? name="xs:QName"?
1967             requires="list of xs:QName" policySets="list of
1968 xs:QName"?/>*
1969         <callback>?
1970             <binding uri="xs:anyURI"? name="xs:QName"?
1971                 requires="list of xs:QName"?
1972                 policySets="list of xs:QName"?/>+
1973         </callback>
1974     </reference>
1975 </component>
1976
1977     ...

```


1978
1979 `</composite>`

1980
1981
1982 The implementation.composite element has the following attribute:

- 1983
 - **name (required)** – the name of the composite used as an implementation
- 1984

1985 6.5.1 Example of Composite used as a Component Implementation

1986
1987 The following is an example of a composite which contains two components, each of which is
1988 implemented by a composite:
1989

```
1990 <?xml version="1.0" encoding="UTF-8"?>
1991 <!-- CompositeComponent example -->
1992 <composite xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
1993     xsd:schemaLocation="http://www.oesa.org/xmlns/sca/1.0
1994 file:/C:/Strategy/SCA/v09_osoaschemas/schemas/sca.xsd"
1995     xmlns="http://www.oesa.org/xmlns/sca/1.0"
1996     targetNamespace="http://foo.com"
1997     xmlns:foo="http://foo.com"
1998     name="AccountComposite">
1999
2000     <service name="AccountService" promote="AccountServiceComponent">
2001         <interface.java interface="services.account.AccountService"/>
2002         <binding.ws port="AccountService#
2003             wsdl.endpoint(AccountService/AccountServiceSOAP)"/>
2004     </service>
2005
2006     <reference name="stockQuoteService"
2007         promote="AccountServiceComponent/StockQuoteService">
2008         <interface.java
2009 interface="services.stockquote.StockQuoteService"/>
2010         <binding.ws
2011 port="http://www.quickstockquote.com/StockQuoteService#
2012             wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
2013     </reference>
2014
2015     <property name="currency" type="xsd:string">EURO</property>
2016
2017     <component name="AccountServiceComponent">
2018         <implementation.composite name="foo:AccountServiceCompositel"/>
2019
2020     <reference name="AccountDataService" target="AccountDataService"/>
```

```

2021         <reference name="StockQuoteService"/>
2022
2023         <property name="currency" source="$currency"/>
2024     </component>
2025
2026     <component name="AccountDataService">
2027         <implementation.composite name="foo:AccountDataServiceComposite"/>
2028
2029         <property name="currency" source="$currency"/>
2030     </component>
2031
2032 </composite>
2033

```

2034 6.6 Using Composites through Inclusion

2035 In order to assist team development, composites may be developed in the form of multiple
 2036 physical artifacts that are merged into a single logical unit.

2037 A composite is defined in an *xxx.composite* file and the composite may receive additional
 2038 content through the *inclusion of other composite* files.

2039 The semantics of included composites are that the content of the included composite is inlined into
 2040 the using composite *xxx.composite* file through *include* elements in the using composite. The
 2041 effect is one of *textual inclusion* – that is, the text content of the included composite is placed
 2042 into the using composite in place of the include statement. The included composite element itself
 2043 is discarded in this process – only its contents are included.

2044 The composite file used for inclusion can have any contents, but always contains a single
 2045 *composite* element. The composite element may contain any of the elements which are valid as
 2046 child elements of a composite element, namely components, services, references, wires and
 2047 includes. There is no need for the content of an included composite to be complete, so that
 2048 artifacts defined within the using composite or in another associated included composite file may
 2049 be referenced. For example, it is permissible to have two components in one composite file while a
 2050 wire specifying one component as the source and the other as the target can be defined in a
 2051 second included composite file.

2052 It is an error if the (using) composite resulting from the inclusion is invalid – for example, if there
 2053 are duplicated elements in the using composite (eg. two services with the same uri contributed by
 2054 different included composites), or if there are wires with non-existent source or target.

2055 The following snippet shows the partial schema for the include element.

```

2056
2057 <?xml version="1.0" encoding="UTF-8"?>
2058 <!-- Include snippet -->
2059 <composite      xmlns="http://www.oesa.org/xmlns/sca/1.0"
2060                targetNamespace="xs:anyURI"
2061                name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?
2062                constrainingType="QName"?
2063                requires="list of xs:QName"? policySets="list of
2064 xs:QName"?>
2065
2066     ...

```

```

2067
2068     <include name="xs:QName" />*
2069
2070     ...
2071
2072 </composite>
2073

```

2074 The include element has the following *attribute*:

- 2075 • **name (required)** – the name of the composite that is included.

2076

2077 6.6.1 Included Composite Examples

2078

2079 The following figure shows the assembly diagram for the MyValueComposite2 containing four

2080 included composites. The **MyValueServices composite** contains the MyValueService service. The

2081 **MyValueComponents composite** contains the MyValueServiceComponent and the

2082 StockQuoteMediatorComponent as well as the wire between them. The **MyValueReferences**

2083 **composite** contains the CustomerService and StockQuoteService references. The **MyValueWires**

2084 **composite** contains the wires that connect the MyValueService service to the

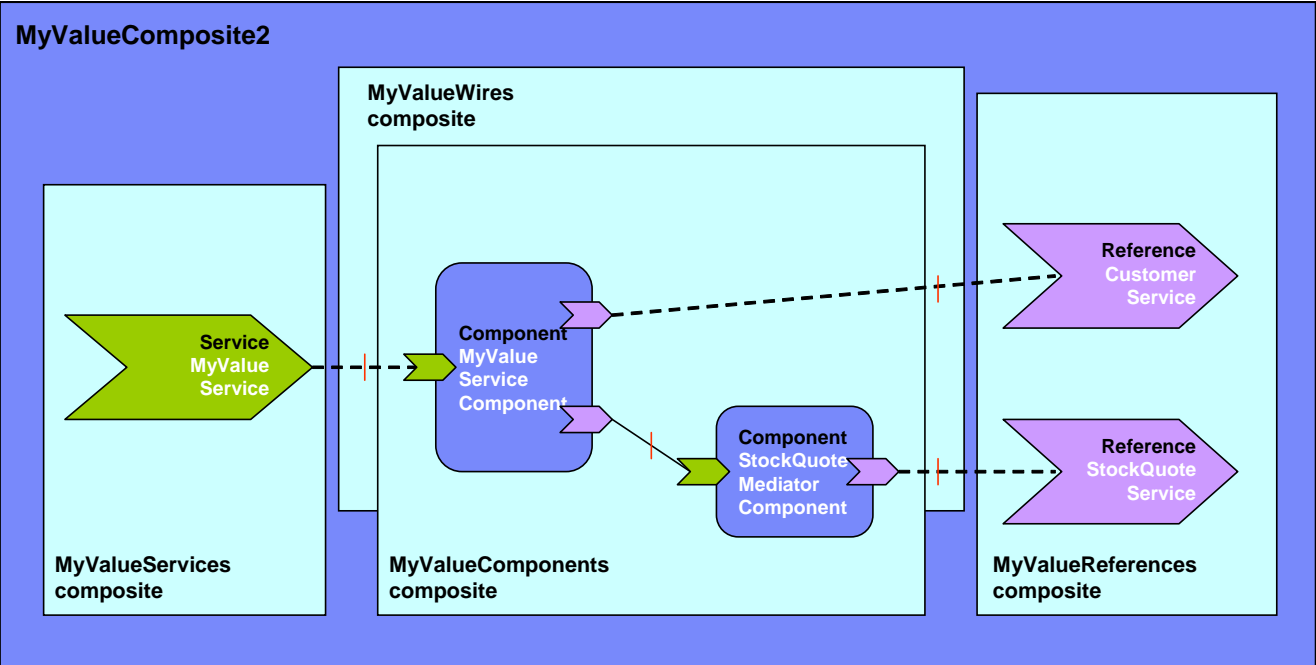
2085 MyValueServiceComponent, that connect the customerService reference of the

2086 MyValueServiceComponent to the CustomerService reference, and that connect the

2087 stockQuoteService reference of the StockQuoteMediatorComponent to the StockQuoteService

2088 reference. Note that this is just one possible way of building the MyValueComposite2 from a set of

2089 included composites.



2090

2091

2092 *Figure 13 MyValueComposite2 built from 4 included composites*

2093

2094 The following snippet shows the contents of the MyValueComposite2.composite file for the

2095 MyValueComposite2 built using included composites. In this sample it only provides the name of

2096 the composite. The composite file itself could be used in a scenario using included composites to
2097 define components, services, references and wires.

2098

```
2099 <?xml version="1.0" encoding="ASCII"?>
2100 <composite xmlns="http://www.oesa.org/xmlns/sca/1.0"
2101           targetNamespace="http://foo.com"
2102           xmlns:foo="http://foo.com"
2103           name="MyValueComposite2" >
2104
2105     <include name="foo:MyValueServices"/>
2106     <include name="foo:MyValueComponents"/>
2107     <include name="foo:MyValueReferences"/>
2108     <include name="foo:MyValueWires"/>
2109
2110 </composite>
```

2111

2112 The following snippet shows the content of the MyValueServices.composite file.

2113

```
2114 <?xml version="1.0" encoding="ASCII"?>
2115 <composite xmlns="http://www.oesa.org/xmlns/sca/1.0"
2116           targetNamespace="http://foo.com"
2117           xmlns:foo="http://foo.com"
2118           name="MyValueServices" >
2119
2120     <service name="MyValueService" promote="MyValueServiceComponent">
2121       <interface.java interface="services.myvalue.MyValueService"/>
2122       <binding.ws port="http://www.myvalue.org/MyValueService#
2123                 wsdl.endpoint(MyValueService/MyValueServiceSOAP)"/>
2124     </service>
2125
2126 </composite>
```

2127

2128 The following snippet shows the content of the MyValueComponents.composite file.

2129

```
2130 <?xml version="1.0" encoding="ASCII"?>
2131 <composite xmlns="http://www.oesa.org/xmlns/sca/1.0"
2132           targetNamespace="http://foo.com"
2133           xmlns:foo="http://foo.com"
2134           name="MyValueComponents" >
2135
2136     <component name="MyValueServiceComponent">
2137       <implementation.java
2138       class="services.myvalue.MyValueServiceImpl"/>

```

```

2139         <property name="currency">EURO</property>
2140     </component>
2141
2142     <component name="StockQuoteMediatorComponent">
2143         <implementation.java class="services.myvalue.SQMediatorImpl"/>
2144         <property name="currency">EURO</property>
2145     </component>
2146
2147 <composite>
2148

```

2149 The following snippet shows the content of the MyValueReferences.composite file.

```

2150
2151 <?xml version="1.0" encoding="ASCII"?>
2152 <composite xmlns="http://www.oesa.org/xmlns/sca/1.0"
2153     targetNamespace="http://foo.com"
2154     xmlns:foo="http://foo.com"
2155     name="MyValueReferences" >
2156
2157     <reference name="CustomerService"
2158         promote="MyValueServiceComponent/CustomerService">
2159         <interface.java interface="services.customer.CustomerService"/>
2160         <binding.sca/>
2161     </reference>
2162
2163     <reference name="StockQuoteService"
2164 promote="StockQuoteMediatorComponent">
2165         <interface.java
2166 interface="services.stockquote.StockQuoteService"/>
2167         <binding.ws port="http://www.stockquote.org/StockQuoteService#
2168             wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
2169     </reference>
2170
2171 </composite>

```

2172 The following snippet shows the content of the MyValueWires.composite file.

```

2173
2174 <?xml version="1.0" encoding="ASCII"?>
2175 <composite xmlns="http://www.oesa.org/xmlns/sca/1.0"
2176     targetNamespace="http://foo.com"
2177     xmlns:foo="http://foo.com"
2178     name="MyValueWires" >
2179
2180     <wire source="MyValueServiceComponent/stockQuoteService"
2181         target="StockQuoteMediatorComponent"/>

```

2182
2183 </composite>

2184 **6.7 Composites which Include Component Implementations of** 2185 **Multiple Types**

2186
2187 A Composite containing multiple components MAY have multiple component implementation types.
2188 For example, a Composite may include one component with a Java POJO as its implementation
2189 and another component with a BPEL process as its implementation.

2190

2191 **6.8 ConstrainingType**

2192 SCA allows a component, and its associated implementation, to be constrained by a
2193 **constrainingType**. The **constrainingType** element provides assistance in developing top-down
2194 usecases in SCA, where an architect or assembler can define the structure of a composite,
2195 including the required form of component implementations, before any of the implementations are
2196 developed.

2197 A **constrainingType** is expressed as an element which has services, reference and properties as
2198 child elements and which can have intents applied to it. The **constrainingType** is independent of
2199 any implementation. Since it is independent of an implementation it cannot contain any
2200 implementation-specific configuration information or defaults. Specifically, it cannot contain
2201 bindings, policySets, property values or default wiring information. The **constrainingType** is
2202 applied to a component through a **constrainingType** attribute on the component.

2203 A **constrainingType** provides the "shape" for a component and its implementation. Any component
2204 configuration that points to a **constrainingType** is constrained by this shape. The **constrainingType**
2205 specifies the services, references and properties that must be implemented. This provides the
2206 ability for the implementer to program to a specific set of services, references and properties as
2207 defined by the **constrainingType**. Components are therefore configured instances of
2208 implementations and are constrained by an associated **constrainingType**.

2209 If the configuration of the component or its implementation do not conform to the
2210 **constrainingType**, it is an error.

2211 A **constrainingType** is represented by a **constrainingType** element. The following snippet shows
2212 the pseudo-schema for the composite element.

2213

```
2214     <?xml version="1.0" encoding="ASCII"?>  
2215     <!-- ConstrainingType schema snippet -->  
2216     <constrainingType     xmlns="http://www.oesa.org/xmlns/sca/1.0"  
2217                         targetNamespace="xs:anyURI"?  
2218                         name="xs:NCName" requires="list of xs:QName"?>
```

2219

2220

```
2221         <service name="xs:NCName" requires="list of xs:QName"?>*  
2222             <interface/?>  
2223         </service>
```

2224

```
2225         <reference name="xs:NCName"  
2226                     multiplicity="0..1 or 1..1 or 0..n or 1..n"?  
2227                     requires="list of xs:QName"?>*
```

```

2228         <interface/>?
2229     </reference>
2230
2231     <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
2232         many="xs:boolean"? mustSupply="xs:boolean"?>*
2233         default-property-value?
2234     </property>
2235
2236 </constrainingType>
2237

```

2238 The constrainingType element has the following *attributes*:

- 2239 • **name (required)** – the name of the constrainingType. The form of a constrainingType
2240 name is an XML QName, in the namespace identified by the targetNamespace attribute.
- 2241 • **targetNamespace (optional)** – an identifier for a target namespace into which the
2242 constrainingType is declared
- 2243 • **requires (optional)** – a list of policy intents. See [the Policy Framework specification \[10\]](#)
2244 for a description of this attribute.

2245 ConstrainingType contains *zero or more properties, services, references*.

2246

2247 When an implementation is constrained by a constrainingType it must define all the services,
2248 references and properties specified in the corresponding constrainingType. The constraining type's
2249 references and services will have interfaces specified and may have intents specified. An
2250 implementation may contain additional services, additional optional references and additional
2251 optional properties, but cannot contain additional non-optional references or additional non-
2252 optional properties (a non-optional property is one with no default value applied).

2253 When a component is constrained by a constrainingType (via the "constrainingType" attribute),
2254 the entire componentType associated with the component and its implementation is not visible to
2255 the containing composite. The containing composite can only see a projection of the
2256 componentType associated with the component and implementation as scoped by the
2257 constrainingType of the component. For example, an additional service provided by the
2258 implementation which is not in the constrainingType associated with the component cannot be
2259 promoted by the containing composite. This requirement ensures that the constrainingType
2260 contract cannot be violated by the composite.

2261 The constrainingType can include required intents on any element. Those intents are applied to
2262 any component that uses that constrainingType. In other words, if requires="reliability" exists on
2263 a constrainingType, or its child service or reference elements, then a constrained component or its
2264 implementation must include requires="reliability" on the component or implementation or on its
2265 corresponding service or reference. Note that the component or implementation may use a
2266 qualified form of an intent specified in unqualified form in the constrainingType, but if the
2267 constrainingType uses the qualified form, then the component or implementation must also use
2268 the qualified form, otherwise there is an error.

2269 A constrainingType can be applied to an implementation. In this case, the implementation's
2270 componentType has a constrainingType attribute set to the QName of the constrainingType.

2271

2272 6.8.1 Example constrainingType

2273

2274 The following snippet shows the contents of the component called "MyValueServiceComponent"
2275 which is constrained by the constrainingType myns:CT. The componentType associated with the
2276 implementation is also shown.

```
2277
2278     <component name="MyValueServiceComponent" constrainingType="myns:CT">
2279         <implementation.java class="services.myvalue.MyValueServiceImpl"/>
2280         <property name="currency">EURO</property>
2281         <reference name="customerService" target="CustomerService">
2282             <binding.ws ...>
2283         <reference name="StockQuoteService"
2284             target="StockQuoteMediatorComponent"/>
2285     </component>
2286
2287     <constrainingType name="CT"
2288         targetNamespace="http://myns.com">
2289         <service name="MyValueService">
2290             <interface.java interface="services.myvalue.MyValueService"/>
2291         </service>
2292         <reference name="customerService">
2293             <interface.java interface="services.customer.CustomerService"/>
2294         </reference>
2295         <reference name="stockQuoteService">
2296             <interface.java interface="services.stockquote.StockQuoteService"/>
2297         </reference>
2298         <property name="currency" type="xsd:string"/>
2299     </constrainingType>
```

2300 The component MyValueServiceComponent is constrained by the constrainingType CT which
2301 means that it must provide:

- 2302 • service **MyValueService** with the interface services.myvalue.MyValueService
- 2303 • reference **customerService** with the interface services.stockquote.StockQuoteService
- 2304 • reference **stockQuoteService** with the interface services.stockquote.StockQuoteService
- 2305 • property **currency** of type xsd:string.

2306

7 Binding

2307

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).

2308

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.

2309
2310

2311

2312

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.

2313

2314

2315

2316

2317

2318

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.

2319

2320

2321

2322

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

2323

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

2324

```
<composite xmlns="http://www.oesa.org/xmlns/sca/1.0"
```

2325

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

2326

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

2327

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

2328

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

2329

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

2330

2331

```
...
```

2332

2333

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

2334

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

2335

```
<interface/>?
```

2336

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

2337

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

2338

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

2339

```
<callback>?
```

2340

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

2341

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

2342

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

2343

```
</callback>
```

2344

```
</service>
```

2345

2346

```
...
```

2347

2348

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

```

2349     promote="list of xs:anyURI"? wiredByImpl="xs:boolean"?
2350     multiplicity="0..1 or 1..1 or 0..n or 1..n"?
2351     requires="list of xs:QName"? policySets="list of xs:QName"?>*
2352     <interface/>?
2353     <binding uri="xs:anyURI"? name="xs:QName"?
2354         requires="list of xs:QName"? policySets="list of
2355 xs:QName"?/>*
2356     <callback>?
2357         <binding uri="xs:anyURI"? name="xs:QName"?
2358             requires="list of xs:QName"?
2359             policySets="list of xs:QName"?/>+
2360     </callback>
2361 </reference>
2362
2363     ...
2364
2365 </composite>
2366

```

2367 The element name of the binding element is architected; it is in itself a qualified name. The first
2368 qualifier is always named "binding", and the second qualifier names the respective binding-type
2369 (e.g. binding.composite, binding.ws, binding.ejb, binding.eis).

2370

2371 A binding element has the following attributes:

- 2372
- 2373 • **uri (optional)** - has the following semantic.
 - 2374 ○ For a binding of a **reference** the URI attribute defines the target URI of the
 - 2375 reference (either the component/service for a wire to an endpoint within the SCA
 - 2376 domain or the accessible address of some endpoint outside the SCA domain). It is
 - 2377 optional for references defined in composites used as component implementations,
 - 2378 but required for references defined in composites contributed to SCA domains. The
 - 2379 URI attribute of a reference of a composite can be reconfigured by a component in
 - 2380 a containing composite using the composite as an implementation. Some binding
 - 2381 types may require that the address of the target service uses more than a simple
 - 2382 URI (such as a WS-Addressing endpoint reference). In those cases, the binding
 - 2383 type will define the additional attributes or sub-elements that are necessary to
 - 2384 identify the service.
 - 2385 ○ For a binding of a **service** the URI attribute defines the URI relative to the
 - 2386 component which contributes the service to the SCA domain. The default value for
 - 2387 the URI is the the value of the name attribute of the binding.
 - 2387 • **name (optional)** – a name for the binding instance (a QName). The name attribute
 - 2388 allows distinction between multiple binding elements on a single service or reference. The
 - 2389 default value of the name attribute is the service or reference name. When a service or
 - 2390 reference has multiple bindings, only one can have the default value; all others must have
 - 2391 a value specified that is unique within the service or reference. The name also permits the
 - 2392 binding instance to be referenced from elsewhere – particularly useful for some types of
 - 2393 binding, which can be declared in a definitions document as a template and referenced
 - 2394 from other binding instances, simplifying the definition of more complex binding instances
 - 2395 (see [the JMS Binding specification \[11\]](#) for examples of this referencing).
 - 2396 • **requires (optional)** - a list of policy intents. See the [Policy Framework specification \[10\]](#)
 - 2397 for a description of this attribute.

- 2398 • **policySets (optional)** – a list of policy sets. See the [Policy Framework specification \[10\]](#)
2399 for a description of this attribute.

2400 When multiple bindings exist for an service, it means that the service is available by any of the
2401 specified bindings. The technique that the SCA runtime uses to choose among available bindings
2402 is left to the implementation and it may include additional (nonstandard) configuration. Whatever
2403 technique is used SHOULD be documented.

2404 Services and References can always have their bindings overridden at the SCA domain level,
2405 unless restricted by Intents applied to them.

2406 The following sections describe the SCA and Web service binding type in detail.

2407

2408 7.1 Messages containing Data not defined in the Service Interface

2409

2410 It is possible for a message to include information that is not defined in the interface used to
2411 define the service, for instance information may be contained in SOAP headers or as MIME
2412 attachments.

2413 Implementation types MAY make this information available to component implementations in their
2414 execution context. These implementation types must indicate how this information is accessed
2415 and in what form they are presented.

2416

2417 7.2 Form of the URI of a Deployed Binding

2418

2419 7.2.1 Constructing Hierarchical URIs

2420 Bindings that use hierarchical URI schemes construct the effective URI with a combination of the
2421 following pieces:

2422 Base System URI for a scheme / Component URI / Service Binding URI

2423

2424 Each of these components deserves addition definition:

2425 **Base Domain URI for a scheme.** An SCA domain should define a base URI for each hierarchical
2426 URI scheme on which it intends to provide services.

2427 For example: the HTTP and HTTPS schemes would each have their own base URI defined for the
2428 domain. An example of a scheme that is not hierarchical, and therefore will have no base URI is
2429 the "jms:" scheme.

2430 **Component URI.** The component URI above is for a component that is deployed in the SCA
2431 Domain. The URI of a component defaults to the name of the component, which is used as a
2432 relative URI. The component may have a specified URI value. The specified URI value may be an
2433 absolute URI in which case it becomes the Base URI for all the services belonging to the
2434 component. If the specified URI value is a relative URI, it is used as the Component URI value
2435 above.

2436 **Service Binding URI.** The Service Binding URI is the relative URI specified in the "uri" attribute
2437 of a binding element of the service. The default value of the attribute is value of the binding's
2438 name attribute treated as a relative URI. If multiple bindings for a single service use the same
2439 scheme (e.g. HTTP), then only one of the bindings may depend on the default value for the uri
2440 attribute, i.e. only one may use the default binding name. The service binding URI may also be
2441 absolute, in which case the absolute URI fully specifies the full URI of the service. Some
2442 deployment environments may not support the use of absolute URIs in service bindings.

2443 Where a component has only a single service, the default value of the Service Binding URI is null,
2444 so that the effective URI is:

2445 Base Domain URI for a scheme / Component URI

2446 This shortened form of the URI is consistent with the shortened form for the wire target URI used
2447 when wiring references to services

2448 Services deployed into the Domain (as opposed to services of components) have a URI that does
2449 not include a component name, i.e.:

2450 Base Domain URI for a scheme / Service Binding URI

2451 The name of the containing composite does not contribute to the URI of any service.

2452 For example, a service where the Base URI is "http://acme.com", the component is named
2453 "stocksComponent" and the service binding name is "getQuote", the URI would look like this:

2454 http://acme.com/stocksComponent/getQuote

2455 Allowing a binding's relative URI to be specified that differs from the name of the service allows
2456 the URI hierarchy of services to be designed independently of the organization of the domain.

2457 It is good practice to design the URI hierarchy to be independent of the domain organization, but
2458 there may be times when domains are initially created using the default URI hierarchy. When this
2459 is the case, the organization of the domain can be changed, while maintaining the form of the URI
2460 hierarchy, by giving appropriate values to the *uri* attribute of select elements. Here is an example
2461 of a change that can be made to the organization while maintaining the existing URIs:

2462 To move a subset of the services out of one component (say "foo") to a new component (say
2463 "bar"), the new component should have bindings for the moved services specify a URI
2464 "../foo/MovedService"..

2465 The URI attribute may also be used in order to create shorter URIs for some endpoints, where the
2466 component name may not be present in the URI at all. For example, if a binding has a *uri*
2467 attribute of "../myService" the component name will not be present in the URI.

2468 7.2.2 Non-hierarchical URIs

2469 Bindings that use non-hierarchical URI schemes (such as jms: or mailto:) may optionally make
2470 use of the "uri" attribute, which is the complete representation of the URI for that service
2471 binding. Where the binding does not use the "uri" attribute, the binding must offer a different
2472 mechanism for specifying the service address.

2473 7.2.3 Determining the URI scheme of a deployed binding

2474 One of the things that needs to be determined when building the effective URI of a deployed
2475 binding (i.e. endpoint) is the URI scheme. The process of determining the endpoint URI scheme is
2476 binding type specific.

2477 If the binding type supports a single protocol then there is only one URI scheme associated with it.
2478 In this case, that URI scheme is used.

2479 If the binding type supports multiple protocols, the binding type implementation determines the
2480 URI scheme by introspecting the binding configuration, which may include the policy sets
2481 associated with the binding.

2482 A good example of a binding type that supports multiple protocols is binding.ws, which can be
2483 configured by referencing either an "abstract" WSDL element (i.e. portType or interface) or a
2484 "concrete" WSDL element (i.e. binding, port or endpoint). When the binding references a PortType
2485 or Interface, the protocol and therefore the URI scheme is derived from the intents/policy sets
2486 attached to the binding. When the binding references a "concrete" WSDL element, there are two
2487 cases:

2488 1) The referenced WSDL binding element uniquely identifies a URI scheme. This is the most
2489 common case. In this case, the URI scheme is given by the protocol/transport specified in the
2490 WSDL binding element.

2491 2) The referenced WSDL binding element doesn't uniquely identify a URI scheme. For example,
2492 when HTTP is specified in the @transport attribute of the SOAP binding element, both "http"
2493 and "https" could be used as valid URI schemes. In this case, the URI scheme is determined
2494 by looking at the policy sets attached to the binding.

2495 It's worth noting that an intent supported by a binding type may completely change the behavior
2496 of the binding. For example, when the intent "confidentiality/transport" is required by an HTTP
2497 binding, SSL is turned on. This basically changes the URI scheme of the binding from "http" to
2498 "https".

2499

2500 7.3 SCA Binding

2501 The SCA binding element is defined by the following schema.

2502

```
2503 <binding.sca />
```

2504

2505 The SCA binding can be used for service interactions between references and services contained
2506 within the SCA domain. The way in which this binding type is implemented is not defined by the
2507 SCA specification and it can be implemented in different ways by different SCA runtimes. The only
2508 requirement is that the required qualities of service must be implemented for the SCA binding
2509 type. The SCA binding type is *not* intended to be an interoperable binding type. For
2510 interoperability, an interoperable binding type such as the Web service binding should be used.

2511 A service or reference definition with no binding element specified uses the SCA binding.
2512 <binding.sca/> would only have to be specified in override cases, or when you specify a
2513 set of bindings on a service or reference definition and the SCA binding should be one of
2514 them.

2515

2516 If the interface of the service or reference is local, then the local variant of the SCA
2517 binding will be used. If the interface of the service or reference is remotable, then either
2518 the local or remote variant of the SCA binding will be used depending on whether source
2519 and target are co-located or not.

2520 If a reference specifies an URI via its uri attribute, then this provides the default wire to a service
2521 provided by another domain level component. The value of the URI has to be as follows:

- 2522 • <domain-component-name>/<service-name>

2523

2524 7.3.1 Example SCA Binding

2525 The following snippet shows the MyValueComposite.composite file for the MyValueComposite
2526 containing the service element for the MyValueService and a reference element for the
2527 StockQuoteService. Both the service and the reference use an SCA binding. The target for the
2528 reference is left undefined in this binding and would have to be supplied by the composite in which
2529 this composite is used.

2530

```
2531 <?xml version="1.0" encoding="ASCII"?>
2532 <!-- Binding SCA example -->
2533 <composite xmlns="http://www.oesa.org/xmlns/sca/1.0"
2534           targetNamespace="http://foo.com"
2535           name="MyValueComposite" >
```

```
2536
2537     <service name="MyValueService" promote="MyValueComponent">
2538         <interface.java interface="services.myvalue.MyValueService"/>
2539         <binding.sca/>
2540         ...
2541     </service>
2542
2543     ...
2544
2545     <reference name="StockQuoteService"
2546 promote="MyValueComponent/StockQuoteReference">
2547         <interface.java
2548 interface="services.stockquote.StockQuoteService"/>
2549         <binding.sca/>
2550     </reference>
2551
2552 </composite>
2553
```

2554 **7.4 Web Service Binding**

2555 SCA defines a Web services binding. This is described in [a separate specification document \[9\]](#).

2556

2557 **7.5 JMS Binding**

2558 SCA defines a JMS binding. This is described in [a separate specification document \[11\]](#).

2559

8 SCA Definitions

2560
2561
2562

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

2563
2564
2565

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

2566

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

2567

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

2568

```
<definitions xmlns="http://www.oesa.org/xmlns/sca/1.0"
```

2569

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

2570

2571

```
<sca:intent/*
```

2572

2573

```
<sca:policySet/*
```

2574

2575

```
<sca:binding/*
```

2576

2577

```
<sca:bindingType/*
```

2578

2579

```
<sca:implementationType/*
```

2580

2581

```
</definitions>
```

2582

The definitions element has the following attribute:

2583

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

2584

2585

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

2586

2587

2588

2589

2590

2591

9 Extension Model

2592

2593 The assembly model can be extended with support for new interface types, implementation types
2594 and binding types. The extension model is based on XML schema substitution groups. There are
2595 three XML Schema substitution group heads defined in the SCA namespace: **interface**,
2596 **implementation** and **binding**, for interface types, implementation types and binding types,
2597 respectively.

2598 The SCA Client and Implementation specifications and the SCA Bindings specifications (see [1])
2599 use these XML Schema substitution groups to define some basic types of interfaces,
2600 implementations and bindings, but other types can be defined as required, where support for
2601 these extra ones is available from the runtime. The interface type elements, implementation type
2602 elements, and binding type elements defined by the SCA specifications (see [1]) are all part of the
2603 SCA namespace ("http://www.oesa.org/xmlns/sca/1.0"), as indicated in their respective schemas.
2604 New interface types, implementation types and binding types that are defined using this
2605 extensibility model, which are not part of these SCA specifications must be defined in namespaces
2606 other than the SCA namespace.

2607 The "." notation is used in naming elements defined by the SCA specifications (e.g.
2608 <implementation.java ... />, <interface.wsdl ... />, <binding.ws ... />), not as a parallel
2609 extensibility approach but as a naming convention that improves usability of the SCA assembly
2610 language.

2611

2612 **Note:** How to contribute SCA model extensions and their runtime function to an SCA runtime will
2613 be defined by a future version of the specification.

2614

9.1 Defining an Interface Type

2615 The following snippet shows the base definition for the **interface** element and **Interface** type
2616 contained in **sca-core.xsd**; see appendix for complete schema.

2617

```
2618
2619 <?xml version="1.0" encoding="UTF-8"?>
2620 <!-- (c) Copyright SCA Collaboration 2006 -->
2621 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2622         targetNamespace="http://www.oesa.org/xmlns/sca/1.0"
2623         xmlns:sca="http://www.oesa.org/xmlns/sca/1.0"
2624         elementFormDefault="qualified">
2625
2626     ...
2627
2628     <element name="interface" type="sca:Interface" abstract="true"/>
2629     <complexType name="Interface"/>
2630
2631     ...
2632
2633 </schema>
```

2634 In the following snippet we show how the base definition is extended to support Java interfaces.
2635 The snippet shows the definition of the *interface.java* element and the *JavaInterface* type
2636 contained in *sca-interface-java.xsd*.

2637

```
2638 <?xml version="1.0" encoding="UTF-8"?>
2639 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2640         targetNamespace="http://www.oesa.org/xmlns/sca/1.0"
2641         xmlns:sca="http://www.oesa.org/xmlns/sca/1.0">
2642
2643     <element name="interface.java" type="sca:JavaInterface"
2644             substitutionGroup="sca:interface"/>
2645     <complexType name="JavaInterface">
2646         <complexContent>
2647             <extension base="sca:Interface">
2648                 <attribute name="interface" type="NCName"
2649 use="required"/>
2650             </extension>
2651         </complexContent>
2652     </complexType>
2653 </schema>
```

2654 In the following snippet we show an example of how the base definition can be extended by other
2655 specifications to support a new interface not defined in the SCA specifications. The snippet shows
2656 the definition of the *my-interface-extension* element and the *my-interface-extension-type*
2657 type.

```
2658 <?xml version="1.0" encoding="UTF-8"?>
2659 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2660         targetNamespace="http://www.example.org/myextension"
2661         xmlns:sca="http://www.oesa.org/xmlns/sca/1.0"
2662         xmlns:tns="http://www.example.org/myextension">
2663
2664     <element name="my-interface-extension" type="tns:my-interface-
2665 extension-type"
2666             substitutionGroup="sca:interface"/>
2667     <complexType name="my-interface-extension-type">
2668         <complexContent>
2669             <extension base="sca:Interface">
2670                 ...
2671             </extension>
2672         </complexContent>
2673     </complexType>
2674 </schema>
```

2676 9.2 Defining an Implementation Type

2677 The following snippet shows the base definition for the *implementation* element and
2678 *Implementation* type contained in *sca-core.xsd*; see appendix for complete schema.

```
2679  
2680 <?xml version="1.0" encoding="UTF-8"?>  
2681 <!-- (c) Copyright SCA Collaboration 2006 -->  
2682 <schema xmlns="http://www.w3.org/2001/XMLSchema"  
2683         targetNamespace="http://www.osoa.org/xmlns/sca/1.0"  
2684         xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"  
2685         elementFormDefault="qualified">  
2686  
2687     ...  
2688  
2689     <element name="implementation" type="sca:Implementation"  
2690     abstract="true"/>  
2691     <complexType name="Implementation"/>  
2692  
2693     ...  
2694  
2695 </schema>
```

2696
2697 In the following snippet we show how the base definition is extended to support Java
2698 implementation. The snippet shows the definition of the *implementation.java* element and the
2699 *JavaImplementation* type contained in *sca-implementation-java.xsd*.

```
2700  
2701 <?xml version="1.0" encoding="UTF-8"?>  
2702 <schema xmlns="http://www.w3.org/2001/XMLSchema"  
2703         targetNamespace="http://www.osoa.org/xmlns/sca/1.0"  
2704         xmlns:sca="http://www.osoa.org/xmlns/sca/1.0">  
2705  
2706     <element name="implementation.java" type="sca:JavaImplementation"  
2707             substitutionGroup="sca:implementation"/>  
2708     <complexType name="JavaImplementation">  
2709         <complexContent>  
2710             <extension base="sca:Implementation">  
2711                 <attribute name="class" type="NCName"  
2712 use="required"/>  
2713             </extension>  
2714         </complexContent>  
2715     </complexType>  
2716 </schema>
```

2717 In the following snippet we show an example of how the base definition can be extended by other
2718 specifications to support a new implementation type not defined in the SCA specifications. The
2719 snippet shows the definition of the *my-impl-extension* element and the *my-impl-extension-*
2720 *type* type.

```

2721 <?xml version="1.0" encoding="UTF-8"?>
2722 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2723         targetNamespace="http://www.example.org/myextension"
2724         xmlns:sca="http://www.oesa.org/xmlns/sca/1.0"
2725         xmlns:tns="http://www.example.org/myextension">
2726
2727     <element name="my-impl-extension" type="tns:my-impl-extension-type"
2728             substitutionGroup="sca:implementation"/>
2729     <complexType name="my-impl-extension-type">
2730         <complexContent>
2731             <extension base="sca:Implementation">
2732                 ...
2733             </extension>
2734         </complexContent>
2735     </complexType>
2736 </schema>
2737

```

2738 In addition to the definition for the new implementation instance element, there needs to be an
2739 associated implementationType element which provides metadata about the new implementation
2740 type. The pseudo schema for the implementationType element is shown in the following snippet:

```

2741 <implementationType type="xs:QName"
2742                   alwaysProvides="list of intent xs:QName"
2743                   mayProvide="list of intent xs:QName"/>
2744

```

2745 The implementation type has the following attributes:

- 2746 • **type (required)** – the type of the implementation to which this implementationType
2747 element applies. This is intended to be the QName of the implementation element for the
2748 implementation type, such as "sca:implementation.java"
- 2749 • **alwaysProvides (optional)** – a set of intents which the implementation type always
2750 provides. See [the Policy Framework specification \[10\]](#) for details.
- 2751 • **mayProvide (optional)** – a set of intents which the implementation type may provide.
2752 See [the Policy Framework specification \[10\]](#) for details.

2753

2754 9.3 Defining a Binding Type

2755 The following snippet shows the base definition for the *binding* element and *Binding* type
2756 contained in *sca-core.xsd*; see appendix for complete schema.

```

2757
2758 <?xml version="1.0" encoding="UTF-8"?>
2759 <!-- binding type schema snippet -->
2760 <!-- (c) Copyright SCA Collaboration 2006, 2007 -->
2761 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2762         targetNamespace="http://www.oesa.org/xmlns/sca/1.0"
2763         xmlns:sca="http://www.oesa.org/xmlns/sca/1.0"
2764         elementFormDefault="qualified">

```

```

2765
2766     ...
2767
2768     <element name="binding" type="sca:Binding" abstract="true"/>
2769     <complexType name="Binding">
2770         <attribute name="uri" type="anyURI" use="optional"/>
2771         <attribute name="name" type="NCName" use="optional"/>
2772         <attribute name="requires" type="sca:listOfQNames"
2773 use="optional"/>
2774         <attribute name="policySets" type="sca:listOfQNames"
2775 use="optional"/>
2776     </complexType>
2777
2778     ...
2779
2780 </schema>

```

2781 In the following snippet we show how the base definition is extended to support Web service
2782 binding. The snippet shows the definition of the *binding.ws* element and the
2783 *WebServiceBinding* type contained in *sca-binding-webservice.xsd*.

```

2784
2785 <?xml version="1.0" encoding="UTF-8"?>
2786 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2787     targetNamespace="http://www.oesa.org/xmlns/sca/1.0"
2788     xmlns:sca="http://www.oesa.org/xmlns/sca/1.0">
2789
2790     <element name="binding.ws" type="sca:WebServiceBinding"
2791         substitutionGroup="sca:binding"/>
2792     <complexType name="WebServiceBinding">
2793         <complexContent>
2794             <extension base="sca:Binding">
2795                 <attribute name="port" type="anyURI" use="required"/>
2796             </extension>
2797         </complexContent>
2798     </complexType>
2799 </schema>

```

2800 In the following snippet we show an example of how the base definition can be extended by other
2801 specifications to support a new binding not defined in the SCA specifications. The snippet shows
2802 the definition of the *my-binding-extension* element and the *my-binding-extension-type* type.

```

2803 <?xml version="1.0" encoding="UTF-8"?>
2804 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2805     targetNamespace="http://www.example.org/myextension"
2806     xmlns:sca="http://www.oesa.org/xmlns/sca/1.0"
2807     xmlns:tns="http://www.example.org/myextension">
2808

```

```
2809     <element name="my-binding-extension" type="tns:my-binding-extension-
2810 type"
2811     substitutionGroup="sca:binding"/>
2812 <complexType name="my-binding-extension-type">
2813     <complexContent>
2814         <extension base="sca:Binding">
2815             ...
2816         </extension>
2817     </complexContent>
2818 </complexType>
2819 </schema>
2820
```

2821 In addition to the definition for the new binding instance element, there needs to be an associated
2822 bindingType element which provides metadata about the new binding type. The pseudo schema
2823 for the bindingType element is shown in the following snippet:

```
2824 <bindingType type="xs:QName"
2825     alwaysProvides="list of intent QNames"?
2826     mayProvide = "list of intent QNames"?/>
2827
```

2828 The binding type has the following attributes:

- 2829 • **type (required)** – the type of the binding to which this bindingType element applies.
2830 This is intended to be the QName of the binding element for the binding type, such as
2831 "sca:binding.ws"
- 2832 • **alwaysProvides (optional)** – a set of intents which the binding type always provides.
2833 See [the Policy Framework specification \[10\]](#) for details.
- 2834 • **mayProvide (optional)** – a set of intents which the binding type may provide. See [the](#)
2835 [Policy Framework specification \[10\]](#) for details.

2836

2837 10 Packaging and Deployment

2838 10.1 Domains

2839 An **SCA Domain** represents a complete runtime configuration, potentially distributed over a series
2840 of interconnected runtime nodes.

2841 A single SCA domain defines the boundary of visibility for all SCA mechanisms. For example, SCA
2842 wires can only be used to connect components within a single SCA domain. Connections to
2843 services outside the domain must use binding specific mechanisms for addressing services (such
2844 as WSDL endpoint URIs). Also, SCA mechanisms such as intents and policySets can only be used
2845 in the context of a single domain. In general, external clients of a service that is developed and
2846 deployed using SCA should not be able to tell that SCA was used to implement the service – it is
2847 an implementation detail.

2848 The size and configuration of an SCA Domain is not constrained by the SCA Assembly specification
2849 and is expected to be highly variable. An SCA Domain typically represents an area of business
2850 functionality controlled by a single organization. For example, an SCA Domain may be the whole
2851 of a business, or it may be a department within a business.

2852 As an example, for the accounts department in a business, the SCA Domain might cover all
2853 finance-related functions, and it might contain a series of composites dealing with specific areas of
2854 accounting, with one for Customer accounts and another dealing with Accounts Payable.

2855 An SCA domain has the following:

- 2856 • A virtual domain-level composite whose components are deployed and running
- 2857 • A set of *installed contributions* that contain implementations, interfaces and other artifacts
2858 necessary to execute components
- 2859 • A set of logical services for manipulating the set of contributions and the virtual domain-
2860 level composite.

2861 The information associated with an SCA domain can be stored in many ways, including but not
2862 limited to a specific filesystem structure or a repository.

2863 10.2 Contributions

2864 An SCA domain may require a large number of different artifacts in order to work. These artifacts
2865 include artifacts defined by SCA and other artifacts such as object code files and interface
2866 definition files. The SCA-defined artifact types are all XML documents. The root elements of the
2867 different SCA definition documents are: `composite`, `componentType`, `constrainingType` and
2868 `definitions`. XML artifacts that are not defined by SCA but which may be needed by an SCA
2869 domain include XML Schema documents, WSDL documents, and BPEL documents. SCA
2870 constructs, like other XML-defined constructs, use XML qualified names for their identity (i.e.
2871 namespace + local name).

2872 Non-XML artifacts are also required within an SCA domain. The most obvious examples of such
2873 non-XML artifacts are Java, C++ and other programming language files necessary for component
2874 implementations. Since SCA is extensible, other XML and non-XML artifacts may also be required.

2875 SCA defines an interoperable packaging format for contributions (ZIP), as specified below. This
2876 format is not the only packaging format that an SCA runtime can use. SCA allows many different
2877 packaging formats, but requires that the ZIP format be supported. When using the ZIP format for
2878 deploying a contribution, this specification does not specify whether that format is retained after
2879 deployment. For example, a Java EE based SCA runtime may convert the ZIP package to an EAR
2880 package. SCA expects certain characteristics of any packaging:

- 2881 • It must be possible to present the artifacts of the packaging to SCA as a hierarchy of
2882 resources based off of a single root

- 2883
- A directory resource should exist at the root of the hierarchy named META-INF
- 2884
- A document should exist directly under the META-INF directory named sca-contribution.xml which lists the SCA Composites within the contribution that are runnable.
- 2885
- 2886
- 2887
- 2888
- 2889
- 2890
- 2891
- 2892
- 2893
- 2894
- 2895
- 2896
- The same document also optionally lists namespaces of constructs that are defined within the contribution and which may be used by other contributions
- Optionally, additional elements may exist that list the namespaces of constructs that are needed by the contribution and which must be found elsewhere, for example in other contributions. These optional elements may not be physically present in the packaging, but may be generated based on the definitions and references that are present, or they may not exist at all if there are no unresolved references.
- See the section "SCA Contribution Metadata Document" for details of the format of this file.

2897

2898

2899

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

- 2900
- A filesystem directory
- 2901
- An OSGi bundle
- 2902
- A compressed directory (zip, gzip, etc)
- 2903
- A JAR file (or its variants – WAR, EAR, etc)

2904

2905

2906

2907

2908

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

2909

2910

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

2911

2912 **10.2.1 SCA Artifact Resolution**

2913

2914

2915

2916

2917

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

2918

2919

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

- 2920
- wsdlLocation and schemaLocation attributes in references to WSDL and XSD schema artifacts respectively
- 2921
- OSGi bundle mechanisms for resolving Java class and related resource dependencies
- 2922

2923

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

2924

2925

2926

2927

2928

2929

2930

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

2931 The SCA artifact resolution is likely to be most useful for SCA domains containing heterogeneous
2932 mixtures of contribution, where artifact-related or packaging-related mechanisms are unlikely to
2933 work across different kinds of contribution.

2934 SCA artifact resolution works on the principle that a contribution which needs to use artifacts
2935 defined elsewhere expresses these dependencies using *import* statements in metadata belonging
2936 to the contribution. A contribution controls which artifacts it makes available to other
2937 contributions through *export* statements in metadata attached to the contribution.

2938

2939 10.2.2 SCA Contribution Metadata Document

2940 The contribution optionally contains a document that declares runnable composites, exported
2941 definitions and imported definitions. The document is found at the path of META-INF/sca-
2942 contribution.xml relative to the root of the contribution. Frequently some SCA metadata may
2943 need to be specified by hand while other metadata is generated by tools (such as the <import>
2944 elements described below). To accommodate this, it is also possible to have an identically
2945 structured document at META-INF/sca-contribution-generated.xml. If this document exists (or is
2946 generated on an as-needed basis), it will be merged into the contents of sca-contribution.xml,
2947 with the entries in sca-contribution.xml taking priority if there are any conflicting declarations.
2948

2949 The format of the document is:

```
2950 <?xml version="1.0" encoding="ASCII"?>  
2951 <!-- sca-contribution pseudo-schema -->  
2952 <contribution xmlns=http://www.osoa.org/xmlns/sca/1.0>  
2953  
2954     <deployable composite="xs:QName"/>*  
2955     <import namespace="xs:String" location="xs:AnyURI"?/>*  
2956     <export namespace="xs:String"/>*  
2957  
2958 </contribution>  
2959
```

2960 **deployable element:** Identifies a composite which is a composite within the contribution that is a
2961 composite intended for potential inclusion into the virtual domain-level composite. Other
2962 composites in the contribution are not intended for inclusion but only for use by other composites.
2963 New composites can be created for a contribution after it is installed, by using the [add Deployment](#)
2964 [Composite](#) capability and the add To Domain Level Composite capability.

- 2965 • **composite (required)** – The QName of a composite within the contribution.

2966

2967 **Export element:** A declaration that artifacts belonging to a particular namespace are exported
2968 and are available for use within other contributions. An export declaration in a contribution
2969 specifies a namespace, all of whose definitions are considered to be exported. By default,
2970 definitions are not exported.

2971 The SCA artifact export is useful for SCA domains containing heterogeneous mixtures of
2972 contribution packagings and technologies, where artifact-related or packaging-related mechanisms
2973 are unlikely to work across different kinds of contribution.

- 2974 • **namespace (required)** – For XML definitions, which are identified by QNames, the
2975 namespace should be the namespace URI for the exported definitions. For XML
2976 technologies that define multiple *symbol spaces* that can be used within one namespace
2977 (e.g. WSDL port types are a different symbol space from WSDL bindings), all definitions
2978 from all symbol spaces are exported.

2979

2980 Technologies that use naming schemes other than QNames must use a different export

2981 element from the same substitution group as the the SCA <export> element. The
2982 element used identifies the technology, and may use any value for the namespace that is
2983 appropriate for that technology. For example, <export.java> can be used can be used to
2984 export java definitions, in which case the namespace should be a fully qualified package
2985 name.

2986
2987 **Import element:** Import declarations specify namespaces of definitions that are needed by the
2988 definitions and implementations within the contribution, but which are not present in the
2989 contribution. It is expected that in most cases import declarations will be generated based on
2990 introspection of the contents of the contribution. In this case, the import declarations would be
2991 found in the META-INF/ sca-contribution-generated.xml document.

2992 • **namespace (required)** – For XML definitions, which are identified by QNames, the
2993 namespace should be the namespace URI for the imported definitions. For XML
2994 technologies that define multiple *symbol spaces* that can be used within one namespace
2995 (e.g. WSDL port types are a different symbol space from WSDL bindings), all definitions
2996 from all symbol spaces are imported.

2997
2998 Technologies that use naming schemes other than QNames must use a different import
2999 element from the same substitution group as the the SCA <import> element. The
3000 element used identifies the technology, and may use any value for the namespace that is
3001 appropriate for that technology. For example, <import.java> can be used can be used to
3002 import java definitions, in which case the namespace should be a fully qualified package
3003 name.

3004 • **location (optional)** – a URI to resolve the definitions for this import. SCA makes no
3005 specific requirements for the form of this URI, nor the means by which it is resolved. It
3006 may point to another contribution (through its URI) or it may point to some location
3007 entirely outside the SCA Domain.
3008

3009 It is expected that SCA runtimes may define implementation specific ways of resolving location
3010 information for artifact resolution between contributions. These mechanisms will however usually
3011 be limited to sets of contributions of one runtime technology and one hosting environment.

3012 In order to accommodate imports of artifacts between contributions of disparate runtime
3013 technologies, it is strongly suggested that SCA runtimes honor SCA contribution URIs as location
3014 specification.

3015 SCA runtimes that support contribution URIs for cross-contribution resolution of SCA artifacts
3016 should do so similarly when used as @schemaLocation and @wsdlLocation and other artifact
3017 location specifications.

3018 The order in which the import statements are specified may play a role in this mechanism. Since
3019 definitions of one namespace can be distributed across several artifacts, multiple import
3020 declarations can be made for one namespace.
3021

3022 The location value is only a default, and dependent contributions listed in the call to
3023 installContribution should override the value if there is a conflict. However, the specific
3024 mechanism for resolving conflicts between contributions that define conflicting definitions is
3025 implementation specific.

3026
3027 If the value of the location attribute is an SCA contribution URI, then the contribution packaging
3028 may become dependent on the deployment environment. In order to avoid such a dependency,
3029 dependent contributions should be specified only when deploying or updating contributions as
3030 specified in the section 'Operations for Contributions' below.

3031 10.2.3 Contribution Packaging using ZIP

3032 SCA allows many different packaging formats that SCA runtimes can support, but SCA requires
3033 that all runtimes support the ZIP packaging format for contributions. This format allows that

3034 metadata specified by the section 'SCA Contribution Metadata Document' be present. Specifically,
3035 it may contain a top-level "META-INF" directory and a "META-INF/sca-contribution.xml" file and
3036 there may also be an optional "META-INF/sca-contribution-generated.xml" file in the package. SCA
3037 defined artifacts as well as non-SCA defined artifacts such as object files, WSDL definition, Java
3038 classes may be present anywhere in the ZIP archive,

3039 A up to date definition of the ZIP file format is published by PKWARE in [an Application Note on the](#)
3040 [.ZIP file format \[12\]](#).

3041

3042 **10.3 Installed Contribution**

3043 As noted in the section above, the contents of a contribution should not need to be modified in
3044 order to install and use it within a domain. An *installed contribution* is a contribution with all of
3045 the associated information necessary in order to execute *deployable composites* within the
3046 contribution.

3047 An installed contribution is made up of the following things:

- 3048 • Contribution Packaging – the contribution that will be used as the starting point for
3049 resolving all references
- 3050 • Contribution base URI
- 3051 • Dependent contributions: a set of snapshots of other contributions that are used to resolve
3052 the import statements from the root composite and from other dependent contributions
 - 3053 ○ Dependent contributions may or may not be shared with other installed
3054 contributions.
 - 3055 ○ When the snapshot of any contribution is taken is implementation defined, ranging
3056 from the time the contribution is installed to the time of execution
- 3057 • Deployment-time composites.
3058 These are composites that are added into an installed contribution after it has been
3059 deployed. This makes it possible to provide final configuration and access to
3060 implementations within a contribution without having to modify the contribution. These
3061 are optional, as composites that already exist within the contribution may also be used for
3062 deployment.

3063

3064 Installed contributions provide a context in which to resolve qualified names (e.g. QNames in XML,
3065 fully qualified class names in Java).

3066 If multiple dependent contributions have exported definitions with conflicting qualified names, the
3067 algorithm used to determine the qualified name to use is implementation dependent.
3068 Implementations of SCA may also generate an error if there are conflicting names.

3069

3070 **10.3.1 Installed Artifact URIs**

3071 When a contribution is installed, all artifacts within the contribution are assigned URIs, which are
3072 constructed by starting with the base URI of the contribution and adding the relative URI of each
3073 artifact (recalling that SCA requires that any packaging format be able to offer up its artifacts in a
3074 single hierarchy).

3075

3076 **10.4 Operations for Contributions**

3077 SCA Domains provide the following conceptual functionality associated with contributions
3078 (meaning the function may not be represented as addressable services and also meaning that

3079 equivalent functionality may be provided in other ways). The functionality is optional meaning that
3080 some SCA runtimes may choose not to provide that functionality in any way:

3081 **10.4.1 install Contribution & update Contribution**

3082
3083 Creates or updates an installed contribution with a supplied root contribution, and installed at a
3084 supplied base URI. A supplied dependent contribution list specifies the contributions that should
3085 be used to resolve the dependencies of the root contribution and other dependent contributions.
3086 These override any dependent contributions explicitly listed via the location attribute in the import
3087 statements of the contribution.
3088

3089 SCA follows the simplifying assumption that the use of a contribution for resolving anything also
3090 means that all other exported artifacts can be used from that contribution. Because of this, the
3091 dependent contribution list is just a list of installed contribution URIs. There is no need to specify
3092 what is being used from each one.

3093 Each dependent contribution is also an installed contribution, with its own dependent
3094 contributions. By default these dependent contributions of the dependent contributions (which we
3095 will call *indirect dependent contributions*) are included as dependent contributions of the installed
3096 contribution. However, if a contribution in the dependent contribution list exports any conflicting
3097 definitions with an indirect dependent contribution, then the indirect dependent contribution is not
3098 included (i.e. the explicit list overrides the default inclusion of indirect dependent contributions).
3099 Also, if there is ever a conflict between two indirect dependent contributions, then the conflict
3100 must be resolved by an explicit entry in the dependent contribution list.

3101 Note that in many cases, the dependent contribution list can be generated. In particular, if a
3102 domain is careful to avoid creating duplicate definitions for the same qualified name, then it is
3103 easy for this list to be generated by tooling.

3104 **10.4.2 add Deployment Composite & update Deployment Composite**

3105 Adds or updates a deployment composite using a supplied composite ("composite by value" – a
3106 data structure, not an existing resource in the domain) to the contribution identified by a supplied
3107 contribution URI. The added or updated deployment composite is given a relative URI that
3108 matches the @name attribute of the composite, with a ".composite" suffix. Since all composites
3109 must run within the context of a installed contribution (any component implementations or other
3110 definitions are resolved within that contribution), this functionality makes it possible for the
3111 deployer to create a composite with final configuration and wiring decisions and add it to an
3112 installed contribution without having to modify the contents of the root contribution.

3113 Also, in some use cases, a contribution may include only implementation code (e.g. PHP scripts).
3114 It should then be possible for those to be given component names by a (possibly generated)
3115 composite that is added into the installed contribution, without having to modify the packaging.

3116 **10.4.3 remove Contribution**

3117 Removes the deployed contribution identified by a supplied contribution URI.
3118

3119 **10.5 Use of Existing (non-SCA) Mechanisms for Resolving Artifacts**

3120
3121 For certain types of artifact, there are existing and commonly used mechanisms for referencing a
3122 specific concrete location where the artifact can be resolved.

3123 Examples of these mechanisms include:

- 3124 • For WSDL files, the *@wsdlLocation* attribute is a hint that has a URI value pointing to the
3125 place holding the WSDL itself.

- 3126 • For XSDs, the *@schemaLocation* attribute is a hint which matches the namespace to a
3127 URI where the XSD is found.

3128 **Note:** In neither of these cases is the runtime obliged to use the location hint and the URI does
3129 not have to be dereferenced.

3130 SCA permits the use of these mechanisms. Where present, these mechanisms take precedence
3131 over the SCA mechanisms. However, use of these mechanisms is discouraged because tying
3132 assemblies to addresses in this way makes the assemblies less flexible and prone to errors when
3133 changes are made to the overall SCA Domain.

3134 **Note:** If one of these mechanisms is present, but there is a failure to find the resource indicated
3135 when using the mechanism (eg the URI is incorrect or invalid, say) the SCA runtime MUST raise
3136 an error and MUST NOT attempt to use SCA resolution mechanisms as an alternative.

3137

3138 10.6 Domain-Level Composite

3139 The domain-level composite is a virtual composite, in that it is not defined by a composite
3140 definition document. Rather, it is built up and modified through operations on the domain.
3141 However, in other respects it is very much like a composite, since it contains components, wires,
3142 services and references.

3143 The abstract domain-level functionality for modifying the domain-level composite is as follows,
3144 although a runtime may supply equivalent functionality in a different form:

3145 10.6.1 add To Domain-Level Composite

3146 This functionality adds the composite identified by a supplied URI to the Domain Level Composite.
3147 The supplied composite URI must refer to a composite within a installed contribution. The
3148 composite's installed contribution determines how the composite's artifacts are resolved (directly
3149 and indirectly). The supplied composite is added to the domain composite with semantics that
3150 correspond to the domain-level composite having an <include> statement that references the
3151 supplied composite. All of the composite's components become *top-level* components and the
3152 services become externally visible services (eg. they would be present in a WSDL description of
3153 the domain).

3154 10.6.2 remove From Domain-Level Composite

3155 Removes from the Domain Level composite the elements corresponding to the composite
3156 identified by a supplied composite URI. This means that the removal of the components, wires,
3157 services and references originally added to the domain level composite by the identified
3158 composite.

3159 10.6.3 get Domain-Level Composite

3160 Returns a <composite> definition that has an <include> line for each composite that had been
3161 added to the domain level composite. It is important to note that, in dereferencing the included
3162 composites, any referenced artifacts must be resolved in terms of that installed composite.

3163 10.6.4 get QName Definition

3164 In order to make sense of the domain-level composite (as returned by get Domain-Level
3165 Composite), it must be possible to get the definitions for named artifacts in the included
3166 composites. This functionality takes the supplied URI of an installed contribution (which provides
3167 the context), a supplied qualified name of a definition to look up, and a supplied symbol space (as
3168 a QName, eg wsdl:PortType). The result is a single definition, in whatever form is appropriate for
3169 that definition type.

3170 Note that this, like all the other domain-level operations, is a conceptual operation. Its capabilities
3171 should exist in some form, but not necessarily as a service operation with exactly this signature.

3172

3173

3174 A. XML Schemas

3175 A.1 sca.xsd

```
3176
3177 <?xml version="1.0" encoding="UTF-8"?>
3178 <!-- (c) Copyright SCA Collaboration 2006 -->
3179 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3180         targetNamespace="http://www.oesa.org/xmlns/sca/1.0"
3181         xmlns:sca="http://www.oesa.org/xmlns/sca/1.0">
3182
3183     <include schemaLocation="sca-core.xsd"/>
3184
3185     <include schemaLocation="sca-interface-java.xsd"/>
3186     <include schemaLocation="sca-interface-wsdl.xsd"/>
3187
3188     <include schemaLocation="sca-implementation-java.xsd"/>
3189     <include schemaLocation="sca-implementation-composite.xsd"/>
3190
3191     <include schemaLocation="sca-binding-webservice.xsd"/>
3192     <include schemaLocation="sca-binding-jms.xsd"/>
3193     <include schemaLocation="sca-binding-sca.xsd"/>
3194
3195     <include schemaLocation="sca-definitions.xsd"/>
3196     <include schemaLocation="sca-policy.xsd"/>
3197
3198 </schema>
3199
```

3200 A.2 sca-core.xsd

```
3201
3202 <?xml version="1.0" encoding="UTF-8"?>
3203 <!-- (c) Copyright SCA Collaboration 2006, 2007 -->
3204 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3205         targetNamespace="http://www.oesa.org/xmlns/sca/1.0"
3206         xmlns:sca="http://www.oesa.org/xmlns/sca/1.0"
3207         elementFormDefault="qualified">
3208
3209     <element name="componentType" type="sca:ComponentType"/>
3210     <complexType name="ComponentType">
3211         <sequence>
3212             <element ref="sca:implementation" minOccurs="0" maxOccurs="1"/>

```

```

3213     <choice minOccurs="0" maxOccurs="unbounded">
3214         <element name="service" type="sca:ComponentService" />
3215         <element name="reference" type="sca:ComponentReference"/>
3216         <element name="property" type="sca:Property"/>
3217     </choice>
3218     <any namespace="##other" processContents="lax" minOccurs="0"
3219         maxOccurs="unbounded"/>
3220 </sequence>
3221 <attribute name="constrainingType" type="QName" use="optional"/>
3222 <anyAttribute namespace="##any" processContents="lax"/>
3223 </complexType>
3224
3225 <element name="composite" type="sca:Composite"/>
3226 <complexType name="Composite">
3227     <sequence>
3228         <element name="include" type="anyURI" minOccurs="0"
3229             maxOccurs="unbounded"/>
3230         <choice minOccurs="0" maxOccurs="unbounded">
3231             <element name="service" type="sca:Service"/>
3232             <element name="property" type="sca:Property"/>
3233             <element name="component" type="sca:Component"/>
3234             <element name="reference" type="sca:Reference"/>
3235             <element name="wire" type="sca:Wire"/>
3236         </choice>
3237         <any namespace="##other" processContents="lax" minOccurs="0"
3238             maxOccurs="unbounded"/>
3239     </sequence>
3240     <attribute name="name" type="NCName" use="required"/>
3241     <attribute name="targetNamespace" type="anyURI" use="required"/>
3242     <attribute name="local" type="boolean" use="optional"
3243 default="false"/>
3244     <attribute name="autowire" type="boolean" use="optional"
3245 default="false"/>
3246     <attribute name="constrainingType" type="QName" use="optional"/>
3247     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3248     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3249     <anyAttribute namespace="##any" processContents="lax"/>
3250 </complexType>
3251
3252 <complexType name="Service">
3253     <sequence>
3254         <element ref="sca:interface" minOccurs="0" maxOccurs="1" />
3255         <element name="operation" type="sca:Operation" minOccurs="0"

```

```

3256         maxOccurs="unbounded" />
3257     <choice minOccurs="0" maxOccurs="unbounded">
3258         <element ref="sca:binding" />
3259         <any namespace="##other" processContents="lax"
3260             minOccurs="0" maxOccurs="unbounded" />
3261     </choice>
3262     <element ref="sca:callback" minOccurs="0" maxOccurs="1" />
3263     <any namespace="##other" processContents="lax" minOccurs="0"
3264         maxOccurs="unbounded" />
3265 </sequence>
3266 <attribute name="name" type="NCName" use="required" />
3267 <attribute name="promote" type="anyURI" use="required" />
3268 <attribute name="requires" type="sca:listOfQNames" use="optional" />
3269 <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3270 <anyAttribute namespace="##any" processContents="lax" />
3271 </complexType>
3272
3273 <element name="interface" type="sca:Interface" abstract="true" />
3274 <complexType name="Interface" abstract="true"/>
3275
3276 <complexType name="Reference">
3277     <sequence>
3278         <element ref="sca:interface" minOccurs="0" maxOccurs="1" />
3279         <element name="operation" type="sca:Operation" minOccurs="0"
3280             maxOccurs="unbounded" />
3281         <choice minOccurs="0" maxOccurs="unbounded">
3282             <element ref="sca:binding" />
3283             <any namespace="##other" processContents="lax" />
3284         </choice>
3285         <element ref="sca:callback" minOccurs="0" maxOccurs="1" />
3286         <any namespace="##other" processContents="lax" minOccurs="0"
3287             maxOccurs="unbounded" />
3288     </sequence>
3289     <attribute name="name" type="NCName" use="required" />
3290     <attribute name="target" type="sca:listOfAnyURIs" use="optional"/>
3291     <attribute name="wiredByImpl" type="boolean" use="optional"
3292 default="false"/>
3293     <attribute name="multiplicity" type="sca:Multiplicity"
3294         use="optional" default="1..1" />
3295     <attribute name="promote" type="sca:listOfAnyURIs" use="required" />
3296     <attribute name="requires" type="sca:listOfQNames" use="optional" />
3297     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3298     <anyAttribute namespace="##any" processContents="lax" />

```

```

3299     </complexType>
3300
3301     <complexType name="SCAPropertyBase" mixed="true">
3302         <!-- mixed="true" to handle simple type -->
3303         <sequence>
3304             <any namespace="##any" processContents="lax" minOccurs="0"
3305                 maxOccurs="1" />
3306             <!-- NOT an extension point; This xsd:any exists to accept
3307                 the element-based or complex type property
3308                 i.e. no element-based extension point under "sca:property"
3309 -->
3310         </sequence>
3311     </complexType>
3312
3313     <!-- complex type for sca:property declaration -->
3314     <complexType name="Property" mixed="true">
3315         <complexContent>
3316             <extension base="sca:SCAPropertyBase">
3317                 <!-- extension defines the place to hold default value -->
3318                 <attribute name="name" type="NCName" use="required"/>
3319                 <attribute name="type" type="QName" use="optional"/>
3320                 <attribute name="element" type="QName" use="optional"/>
3321                 <attribute name="many" type="boolean" default="false"
3322                     use="optional"/>
3323                 <attribute name="mustSupply" type="boolean" default="false"
3324                     use="optional"/>
3325                 <anyAttribute namespace="##any" processContents="lax"/>
3326                 <!-- an extension point ; attribute-based only -->
3327             </extension>
3328         </complexContent>
3329     </complexType>
3330
3331     <complexType name="PropertyValue" mixed="true">
3332         <complexContent>
3333             <extension base="sca:SCAPropertyBase">
3334                 <attribute name="name" type="NCName" use="required"/>
3335                 <attribute name="type" type="QName" use="optional"/>
3336                 <attribute name="element" type="QName" use="optional"/>
3337                 <attribute name="many" type="boolean" default="false"
3338                     use="optional"/>
3339                 <attribute name="source" type="string" use="optional"/>
3340                 <attribute name="file" type="anyURI" use="optional"/>
3341                 <anyAttribute namespace="##any" processContents="lax"/>

```

```

3342         <!-- an extension point ; attribute-based only -->
3343     </extension>
3344 </complexContent>
3345 </complexType>
3346
3347 <element name="binding" type="sca:Binding" abstract="true"/>
3348 <complexType name="Binding" abstract="true">
3349     <sequence>
3350         <element name="operation" type="sca:Operation" minOccurs="0"
3351             maxOccurs="unbounded" />
3352     </sequence>
3353     <attribute name="uri" type="anyURI" use="optional"/>
3354     <attribute name="name" type="NCName" use="optional"/>
3355     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3356     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3357 </complexType>
3358
3359 <element name="bindingType" type="sca:BindingType"/>
3360 <complexType name="BindingType">
3361     <sequence minOccurs="0" maxOccurs="unbounded">
3362         <any namespace="##other" processContents="lax" />
3363     </sequence>
3364     <attribute name="type" type="QName" use="required"/>
3365     <attribute name="alwaysProvides" type="sca:listOfQNames"
3366 use="optional"/>
3367     <attribute name="mayProvide" type="sca:listOfQNames" use="optional"/>
3368     <anyAttribute namespace="##any" processContents="lax"/>
3369 </complexType>
3370
3371 <element name="callback" type="sca:Callback"/>
3372 <complexType name="Callback">
3373     <choice minOccurs="0" maxOccurs="unbounded">
3374         <element ref="sca:binding"/>
3375         <any namespace="##other" processContents="lax"/>
3376     </choice>
3377     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3378     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3379     <anyAttribute namespace="##any" processContents="lax"/>
3380 </complexType>
3381
3382 <complexType name="Component">
3383     <sequence>
3384         <element ref="sca:implementation" minOccurs="0" maxOccurs="1"/>

```

```

3385         <choice minOccurs="0" maxOccurs="unbounded">
3386             <element name="service" type="sca:ComponentService"/>
3387             <element name="reference" type="sca:ComponentReference"/>
3388             <element name="property" type="sca:PropertyValue" />
3389         </choice>
3390         <any namespace="##other" processContents="lax" minOccurs="0"
3391             maxOccurs="unbounded"/>
3392     </sequence>
3393     <attribute name="name" type="NCName" use="required"/>
3394     <attribute name="autowire" type="boolean" use="optional"
3395 default="false"/>
3396     <attribute name="constrainingType" type="QName" use="optional"/>
3397     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3398     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3399     <anyAttribute namespace="##any" processContents="lax"/>
3400 </complexType>
3401
3402 <complexType name="ComponentService">
3403     <complexContent>
3404         <restriction base="sca:Service">
3405             <sequence>
3406                 <element ref="sca:interface" minOccurs="0"
3407 maxOccurs="1"/>
3408                 <element name="operation" type="sca:Operation"
3409 minOccurs="0"
3410                 maxOccurs="unbounded" />
3411                 <choice minOccurs="0" maxOccurs="unbounded">
3412                     <element ref="sca:binding"/>
3413                     <any namespace="##other" processContents="lax"
3414                         minOccurs="0" maxOccurs="unbounded"/>
3415                 </choice>
3416                 <element ref="sca:callback" minOccurs="0"
3417 maxOccurs="1"/>
3418                 <any namespace="##other" processContents="lax"
3419 minOccurs="0"
3420                 maxOccurs="unbounded" />
3421             </sequence>
3422             <attribute name="name" type="NCName" use="required"/>
3423             <attribute name="requires" type="sca:listOfQNames"
3424                 use="optional"/>
3425             <attribute name="policySets" type="sca:listOfQNames"
3426                 use="optional"/>
3427             <anyAttribute namespace="##any" processContents="lax"/>
3428         </restriction>

```

```

3429     </complexContent>
3430 </complexType>
3431
3432 <complexType name="ComponentReference">
3433     <complexContent>
3434         <restriction base="sca:Reference">
3435             <sequence>
3436                 <element ref="sca:interface" minOccurs="0"
3437 maxOccurs="1" />
3438                 <element name="operation" type="sca:Operation"
3439 minOccurs="0"
3440                 maxOccurs="unbounded" />
3441                 <choice minOccurs="0" maxOccurs="unbounded">
3442                     <element ref="sca:binding" />
3443                     <any namespace="##other" processContents="lax"
3444 />
3445                 </choice>
3446                 <element ref="sca:callback" minOccurs="0"
3447 maxOccurs="1" />
3448                 <any namespace="##other" processContents="lax"
3449 minOccurs="0"
3450                 maxOccurs="unbounded" />
3451             </sequence>
3452             <attribute name="name" type="NCName" use="required" />
3453             <attribute name="autowire" type="boolean" use="optional"
3454                 default="false"/>
3455             <attribute name="wiredByImpl" type="boolean" use="optional"
3456                 default="false"/>
3457             <attribute name="target" type="sca:listOfAnyURIs"
3458 use="optional"/>
3459             <attribute name="multiplicity" type="sca:Multiplicity"
3460                 use="optional" default="1..1" />
3461             <attribute name="requires" type="sca:listOfQNames"
3462 use="optional"/>
3463             <attribute name="policySets" type="sca:listOfQNames"
3464                 use="optional"/>
3465             <anyAttribute namespace="##any" processContents="lax" />
3466         </restriction>
3467     </complexContent>
3468 </complexType>
3469
3470 <element name="implementation" type="sca:Implementation"
3471     abstract="true" />
3472 <complexType name="Implementation" abstract="true">
3473     <attribute name="requires" type="sca:listOfQNames" use="optional"/>

```



```

3474     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3475 </complexType>
3476
3477 <element name="implementationType" type="sca:ImplementationType"/>
3478 <complexType name="ImplementationType">
3479     <sequence minOccurs="0" maxOccurs="unbounded">
3480         <any namespace="##other" processContents="lax" />
3481     </sequence>
3482     <attribute name="type" type="QName" use="required"/>
3483     <attribute name="alwaysProvides" type="sca:listOfQNames"
3484 use="optional"/>
3485     <attribute name="mayProvide" type="sca:listOfQNames" use="optional"/>
3486     <anyAttribute namespace="##any" processContents="lax"/>
3487 </complexType>
3488
3489 <complexType name="Wire">
3490     <sequence>
3491         <any namespace="##other" processContents="lax" minOccurs="0"
3492 maxOccurs="unbounded"/>
3493     </sequence>
3494     <attribute name="source" type="anyURI" use="required"/>
3495     <attribute name="target" type="anyURI" use="required"/>
3496     <anyAttribute namespace="##any" processContents="lax"/>
3497 </complexType>
3498
3499 <element name="include" type="sca:Include"/>
3500 <complexType name="Include">
3501     <attribute name="name" type="QName"/>
3502     <anyAttribute namespace="##any" processContents="lax"/>
3503 </complexType>
3504
3505 <complexType name="Operation">
3506     <attribute name="name" type="NCName" use="required"/>
3507     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3508     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3509     <anyAttribute namespace="##any" processContents="lax"/>
3510 </complexType>
3511
3512 <element name="constrainingType" type="sca:ConstrainingType"/>
3513 <complexType name="ConstrainingType">
3514     <sequence>
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="name" type="NCName" use="required"/>
3524 <attribute name="targetNamespace" type="anyURI"/>
3525 <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3526 <anyAttribute namespace="##any" processContents="lax"/>
3527 </complexType>
3528
3529
3530 <simpleType name="Multiplicity">
3531     <restriction base="string">
3532         <enumeration value="0..1"/>
3533         <enumeration value="1..1"/>
3534         <enumeration value="0..n"/>
3535         <enumeration value="1..n"/>
3536     </restriction>
3537 </simpleType>
3538
3539 <simpleType name="OverrideOptions">
3540     <restriction base="string">
3541         <enumeration value="no"/>
3542         <enumeration value="may"/>
3543         <enumeration value="must"/>
3544     </restriction>
3545 </simpleType>
3546
3547 <!-- Global attribute definition for @requires to permit use of intents
3548     within WSDL documents -->
3549 <attribute name="requires" type="sca:listOfQNames"/>
3550
3551 <!-- Global attribute definition for @endsConversation to mark operations
3552     as ending a conversation -->
3553 <attribute name="endsConversation" type="boolean" default="false"/>
3554
3555 <simpleType name="listOfQNames">
3556     <list itemType="QName"/>
3557 </simpleType>
3558
3559 <simpleType name="listOfAnyURIs">

```

```
3560     <list itemType="anyURI"/>
3561   </simpleType>
3562
3563 </schema>
```

3564 **A.3 sca-binding-sca.xsd**

```
3565
3566 <?xml version="1.0" encoding="UTF-8"?>
3567 <!-- (c) Copyright SCA Collaboration 2006, 2007 -->
3568 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3569   targetNamespace="http://www.oesa.org/xmlns/sca/1.0"
3570   xmlns:sca="http://www.oesa.org/xmlns/sca/1.0"
3571   elementFormDefault="qualified">
3572
3573   <include schemaLocation="sca-core.xsd"/>
3574
3575   <element name="binding.sca" type="sca:SCABinding"
3576     substitutionGroup="sca:binding"/>
3577   <complexType name="SCABinding">
3578     <complexContent>
3579       <extension base="sca:Binding">
3580         <sequence>
3581           <element name="operation" type="sca:Operation"
3582             minOccurs="0"
3583               maxOccurs="unbounded" />
3584         </sequence>
3585         <attribute name="uri" type="anyURI" use="optional"/>
3586         <attribute name="name" type="QName" use="optional"/>
3587         <attribute name="requires" type="sca:listOfQNames"
3588           use="optional"/>
3589         <attribute name="policySets" type="sca:listOfQNames"
3590           use="optional"/>
3591         <anyAttribute namespace="##any" processContents="lax"/>
3592       </extension>
3593     </complexContent>
3594   </complexType>
3595 </schema>
3596
```

3597 **A.4 sca-interface-java.xsd**

```
3598
3599 <?xml version="1.0" encoding="UTF-8"?>
3600 <!-- (c) Copyright SCA Collaboration 2006 -->
```

```

3601 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3602       targetNamespace="http://www.oesa.org/xmlns/sca/1.0"
3603       xmlns:sca="http://www.oesa.org/xmlns/sca/1.0"
3604       elementFormDefault="qualified">
3605
3606   <include schemaLocation="sca-core.xsd"/>
3607
3608   <element name="interface.java" type="sca:JavaInterface"
3609         substitutionGroup="sca:interface"/>
3610   <complexType name="JavaInterface">
3611     <complexContent>
3612       <extension base="sca:Interface">
3613         <sequence>
3614           <any namespace="##other" processContents="lax"
3615 minOccurs="0"                                maxOccurs="unbounded"/>
3616         </sequence>
3617         <attribute name="interface" type="NCName" use="required"/>
3618         <attribute name="callbackInterface" type="NCName"
3619 use="optional"/>
3620         <anyAttribute namespace="##any" processContents="lax"/>
3621       </extension>
3622     </complexContent>
3623   </complexType>
3624 </schema>
3625

```

3626 A.5 sca-interface-wsdl.xsd

```

3627
3628 <?xml version="1.0" encoding="UTF-8"?>
3629 <!-- (c) Copyright SCA Collaboration 2006 -->
3630 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3631       targetNamespace="http://www.oesa.org/xmlns/sca/1.0"
3632       xmlns:sca="http://www.oesa.org/xmlns/sca/1.0"
3633       elementFormDefault="qualified">
3634
3635   <include schemaLocation="sca-core.xsd"/>
3636
3637   <element name="interface.wsdl" type="sca:WSDLPortType"
3638         substitutionGroup="sca:interface"/>
3639   <complexType name="WSDLPortType">
3640     <complexContent>
3641       <extension base="sca:Interface">
3642         <sequence>
3643           <any namespace="##other" processContents="lax"
3644 minOccurs="0"                                maxOccurs="unbounded"/>

```

```

3645         </sequence>
3646         <attribute name="interface" type="anyURI" use="required"/>
3647         <attribute name="callbackInterface" type="anyURI"
3648 use="optional"/>
3649         <anyAttribute namespace="##any" processContents="lax"/>
3650     </extension>
3651 </complexContent>
3652 </complexType>
3653 </schema>
3654

```

3655 **A.6 sca-implementation-java.xsd**

```

3656
3657 <?xml version="1.0" encoding="UTF-8"?>
3658 <!-- (c) Copyright SCA Collaboration 2006 -->
3659 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3660     targetNamespace="http://www.oesa.org/xmlns/sca/1.0"
3661     xmlns:sca="http://www.oesa.org/xmlns/sca/1.0"
3662     elementFormDefault="qualified">
3663
3664     <include schemaLocation="sca-core.xsd"/>
3665
3666     <element name="implementation.java" type="sca:JavaImplementation"
3667         substitutionGroup="sca:implementation"/>
3668     <complexType name="JavaImplementation">
3669         <complexContent>
3670             <extension base="sca:Implementation">
3671                 <sequence>
3672                     <any namespace="##other" processContents="lax"
3673                         minOccurs="0" maxOccurs="unbounded"/>
3674                 </sequence>
3675                 <attribute name="class" type="NCName" use="required"/>
3676                 <attribute name="requires" type="sca:listOfQNames"
3677 use="optional"/>
3678                 <attribute name="policySets" type="sca:listOfQNames"
3679                     use="optional"/>
3680                 <anyAttribute namespace="##any" processContents="lax"/>
3681             </extension>
3682         </complexContent>
3683     </complexType>
3684 </schema>

```

3685 **A.7 sca-implementation-composite.xsd**

```
3686
3687 <?xml version="1.0" encoding="UTF-8"?>
3688 <!-- (c) Copyright SCA Collaboration 2006 -->
3689 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3690     targetNamespace="http://www.oesa.org/xmlns/sca/1.0"
3691     xmlns:sca="http://www.oesa.org/xmlns/sca/1.0"
3692     elementFormDefault="qualified">
3693
3694     <include schemaLocation="sca-core.xsd"/>
3695     <element name="implementation.composite" type="sca:SCAImplementation"
3696         substitutionGroup="sca:implementation"/>
3697     <complexType name="SCAImplementation">
3698         <complexContent>
3699             <extension base="sca:Implementation">
3700                 <sequence>
3701                     <any namespace="##other" processContents="lax"
3702 minOccurs="0"
3703                         maxOccurs="unbounded"/>
3704                 </sequence>
3705                 <attribute name="name" type="QName" use="required"/>
3706                 <attribute name="requires" type="sca:listOfQNames"
3707 use="optional"/>
3708                 <attribute name="policySets" type="sca:listOfQNames"
3709 use="optional"/>
3710                 <anyAttribute namespace="##any" processContents="lax"/>
3711             </extension>
3712         </complexContent>
3713     </complexType>
3714 </schema>
3715
```

3716 **A.8 sca-definitions.xsd**

```
3717
3718 <?xml version="1.0" encoding="UTF-8"?>
3719 <!-- (c) Copyright SCA Collaboration 2006 -->
3720 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3721     targetNamespace="http://www.oesa.org/xmlns/sca/1.0"
3722     xmlns:sca="http://www.oesa.org/xmlns/sca/1.0"
3723     elementFormDefault="qualified">
3724
3725     <include schemaLocation="sca-core.xsd"/>
3726
```

```
3727     <element name="definitions">
3728         <complexType>
3729             <choice minOccurs="0" maxOccurs="unbounded">
3730                 <element ref="sca:intent"/>
3731                 <element ref="sca:policySet"/>
3732                 <element ref="sca:binding"/>
3733                 <element ref="sca:bindingType"/>
3734                 <element ref="sca:implementationType"/>
3735                 <any namespace="##other" processContents="lax" minOccurs="0"
3736                     maxOccurs="unbounded"/>
3737             </choice>
3738         </complexType>
3739     </element>
3740 </schema>
3741
3742
```

3743 **A.9 sca-binding-webservice.xsd**

3744 Is described in [the SCA Web Services Binding specification \[9\]](#)

3745 **A.10 sca-binding-jms.xsd**

3746 Is described in [the SCA JMS Binding specification \[11\]](#)

3747 **A.11 sca-policy.xsd**

3748 Is described in [the SCA Policy Framework specification \[10\]](#)

3749

B. SCA Concepts

3750

B.1 Binding

3751 **Bindings** are used by services and references. References use bindings to describe the access
3752 mechanism used to call the service to which they are wired. Services use bindings to describe the
3753 access mechanism(s) that clients should use to call the service.

3754 SCA supports multiple different types of bindings. Examples include **SCA service, Web service,**
3755 **stateless session EJB, data base stored procedure, EIS service.** SCA provides an extensibility
3756 mechanism by which an SCA runtime can add support for additional binding types.

3757

3758

B.2 Component

3759 **SCA components** are configured instances of **SCA implementations**, which provide and consume
3760 services. SCA allows many different implementation technologies such as Java, BPEL, C++. SCA defines
3761 an **extensibility mechanism** that allows you to introduce new implementation types. The current
3762 specification does not mandate the implementation technologies to be supported by an SCA run-time,
3763 vendors may choose to support the ones that are important for them. A single SCA implementation may
3764 be used by multiple Components, each with a different configuration.

3765 The Component has a reference to an implementation of which it is an instance, a set of property values,
3766 and a set of service reference values. Property values define the values of the properties of the
3767 component as defined by the component's implementation. Reference values define the services that
3768 resolve the references of the component as defined by its implementation. These values can either be a
3769 particular service of a particular component, or a reference of the containing composite.

3770

B.3 Service

3771 **SCA services** are used to declare the externally accessible services of an **implementation**. For a
3772 composite, a service is typically provided by a service of a component within the composite, or by a
3773 reference defined by the composite. The latter case allows the republication of a service with a new
3774 address and/or new bindings. The service can be thought of as a point at which messages from external
3775 clients enter a composite or implementation.

3776 A service represents an addressable set of operations of an implementation that are designed to be
3777 exposed for use by other implementations or exposed publicly for use elsewhere (eg public Web services
3778 for use by other organizations). The operations provided by a service are specified by an Interface, as
3779 are the operations required by the service client (if there is one). An implementation may contain
3780 multiple services, when it is possible to address the services of the implementation separately.

3781 A service may be provided **as SCA remote services, as Web services, as stateless session EJB's, as**
3782 **EIS services, and so on.** Services use **bindings** to describe the way in which they are published. SCA
3783 provides an **extensibility mechanism** that makes it possible to introduce new binding types for new
3784 types of services.

3785

B.3.1 Remotable Service

3786 A Remotable Service is a service that is designed to be published remotely in a loosely-coupled
3787 SOA architecture. For example, SCA services of SCA implementations can define
3788 implementations of industry-standard web services. Remotable services use pass-by-value
3789 semantics for parameters and returned results.

3790 A service is remotable if it is defined by a WSDL port type or if it defined by a Java interface
3791 marked with the @Remotable annotation.

3792 B.3.2 Local Service

3793 Local services are services that are designed to be only used “locally” by other implementations
3794 that are deployed concurrently in a tightly-coupled architecture within the same operating system
3795 process.

3796 Local services may rely on by-reference calling conventions, or may assume a very fine-grained
3797 interaction style that is incompatible with remote distribution. They may also use technology-
3798 specific data-types.

3799 Currently a service is local only if it defined by a Java interface not marked with the @Remotable
3800 annotation.

3801

3802 B.4 Reference

3803 **SCA references** represent a dependency that an implementation has on a service that is supplied by
3804 some other implementation, where the service to be used is specified through configuration. In other
3805 words, a reference is a service that an implementation may call during the execution of its business
3806 function. References are typed by an interface.

3807 For composites, composite references can be accessed by components within the composite like any
3808 service provided by a component within the composite. Composite references can be used as the targets
3809 of wires from component references when configuring Components.

3810 A composite reference can be used to access a service such as: an SCA service provided by another
3811 SCA composite, a Web service, a stateless session EJB, a data base stored procedure or an EIS service,
3812 and so on. References use **bindings** to describe the access method used to their services. SCA provides
3813 an **extensibility mechanism** that allows the introduction of new binding types to references.

3814

3815 B.5 Implementation

3816 An implementation is concept that is used to describe a piece of software technology such as a Java
3817 class, BPEL process, XSLT transform, or C++ class that is used to implement one or more services in a
3818 service-oriented application. An SCA composite is also an implementation.

3819 Implementations define points of variability including properties that can be set and settable references to
3820 other services. The points of variability are configured by a component that uses the implementation. The
3821 specification refers to the configurable aspects of an implementation as its **componentType**.

3822 B.6 Interface

3823 **Interfaces** define one or more business functions. These business functions are provided by Services
3824 and are used by components through References. Services are defined by the Interface they implement.
3825 SCA currently supports two interface type systems:

- 3826 • Java interfaces
- 3827 • WSDL portTypes

3828

3829 SCA also provides an extensibility mechanism by which an SCA runtime can add support for additional
3830 interface type systems.

3831 Interfaces may be **bi-directional**. A bi-directional service has service operations which must be provided
3832 by each end of a service communication – this could be the case where a particular service requires a
3833 “callback” interface on the client, which is calls during the process of handing service requests from the
3834 client.

3835

3836 B.7 Composite

3837 An SCA composite is the basic unit of composition within an SCA Domain. An **SCA Composite** is an
3838 assembly of Components, Services, References, and the Wires that interconnect them. Composites can
3839 be used to contribute elements to an **SCA Domain**.

3840 A **composite** has the following characteristics:

- 3841 • It may be used as a component implementation. When used in this way, it defines a boundary for
3842 Component visibility. Components may not be directly referenced from outside of the composite
3843 in which they are declared.
- 3844 • It can be used to define a unit of deployment. Composites are used to contribute business logic
3845 artifacts to an SCA domain.

3846

3847 B.8 Composite inclusion

3848 One composite can be used to provide part of the definition of another composite, through the process of
3849 inclusion. This is intended to make team development of large composites easier. Included composites
3850 are merged together into the using composite at deployment time to form a single logical composite.

3851 Composites are included into other composites through <include.../> elements in the using composite.
3852 The SCA Domain uses composites in a similar way, through the deployment of composite files to a
3853 specific location.

3854

3855 B.9 Property

3856 **Properties** allow for the configuration of an implementation with externally set data values. The data
3857 value is provided through a Component, possibly sourced from the property of a containing composite.

3858 Each Property is defined by the implementation. Properties may be defined directly through the
3859 implementation language or through annotations of implementations, where the implementation language
3860 permits, or through a componentType file. A Property can be either a simple data type or a complex data
3861 type. For complex data types, XML schema is the preferred technology for defining the data types.

3862

3863 B.10 Domain

3864 An SCA Domain represents a set of Services providing an area of Business functionality that is controlled
3865 by a single organization. As an example, for the accounts department in a business, the SCA Domain
3866 might cover all finance-related functions, and it might contain a series of composites dealing with specific
3867 areas of accounting, with one for Customer accounts, another dealing with Accounts Payable.

3868 A domain specifies the instantiation, configuration and connection of a set of components, provided via
3869 one or more composite files. The domain, like a composite, also has Services and References. Domains
3870 also contain Wires which connect together the Components, Services and References.

3871

3872 B.11 Wire

3873 **SCA wires** connect **service references** to **services**.

3874 Within a composite, valid wire sources are component references and composite services. Valid wire
3875 targets are component services and composite references.

3876 When using included composites, the sources and targets of the wires don't have to be declared in the
3877 same composite as the composite that contains the wire. The sources and targets can be defined by
3878 other included composites. Targets can also be external to the SCA domain.

3879

3880

3881

3882

C. Acknowledgements

3883 The following individuals have participated in the creation of this specification and are gratefully
3884 acknowledged:

3885 **Participants:**

3886 [Participant Name, Affiliation | Individual Member]

3887 [Participant Name, Affiliation | Individual Member]

3888

D. Non-Normative Text

3890

E. Revision History

3891

[optional; should not be included in OASIS Standards]

3892

Revision	Date	Editor	Changes Made
1	2007-09-24	Anish Karmarkar	Applied the OASIS template + related changes to the Submission

3893

3894

3895