



Service Component Architecture Assembly Model Specification Version 1.1

Committee Draft 02

14th January 2009

Specification URIs:

This Version:

<http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd02.html>
<http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd02.doc>
<http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd02.pdf> (Authoritative)

Previous Version:

Latest Version:

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

Latest Approved Version:

Technical Committee:

[OASIS Service Component Architecture / Assembly \(SCA-Assembly\) TC](#)

Chair(s):

Martin Chapman, Oracle
Mike Edwards, IBM

Editor(s):

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

Related work:

This specification replaces or supercedes:

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

This specification is related to:

- Service Component Architecture Policy Framework Specification Version 1.1

Declared XML Namespace(s):

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

Abstract:

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

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

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

This document describes the SCA Assembly Model, which covers

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

Status:

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

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

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

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

Notices

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

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

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

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

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

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

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

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

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

Table of Contents

Committee Draft 02	1
Notices	4
Table of Contents	5
1 Introduction	8
1.1 Terminology	8
1.2 Normative References	8
1.3 Naming Conventions	9
2 Overview	10
2.1 Diagram used to Represent SCA Artifacts	11
3 Quick Tour by Sample	13
4 Implementation and ComponentType	14
4.1 Component Type	14
4.1.1 Service	15
4.1.2 Reference	16
4.1.3 Property	18
4.1.4 Implementation	19
4.2 Example ComponentType	20
4.3 Example Implementation	21
5 Component	23
5.1 Implementation	24
5.2 Service	25
5.3 Reference	26
5.3.1 Specifying the Target Service(s) for a Reference	28
5.4 Property	30
5.5 Example Component	33
6 Composite	36
6.1 Service	37
6.1.1 Service Examples	39
6.2 Reference	40
6.2.1 Example Reference	42
6.3 Property	44
6.3.1 Property Examples	45
6.4 Wire	47
6.4.1 Wire Examples	49
6.4.2 Autowire	50
6.4.3 Autowire Examples	51
6.5 Using Composites as Component Implementations	54
6.5.1 Example of Composite used as a Component Implementation	54
6.6 Using Composites through Inclusion	55
6.6.1 Included Composite Examples	56
6.7 Composites which Include Component Implementations of Multiple Types	59
6.8 Structural URI of Components	59
7 ConstrainingType	61

7.1	Example constrainingType	62
8	Interface.....	64
8.1	Local and Remotable Interfaces	64
8.2	Bidirectional Interfaces	65
8.3	Conversational Interfaces	66
8.4	Long-running Request-Response Operations	69
8.4.1	Background	69
8.4.2	Definition of "long-running"	69
8.4.3	The asyncInvocation Intent	69
8.4.4	Requirements on Bindings	69
8.4.5	Implementation Type Support	70
8.5	SCA-Specific Aspects for WSDL Interfaces	70
8.6	WSDL Interface Type	71
8.6.1	Example of interface.wsdl	72
9	Binding.....	73
9.1	Messages containing Data not defined in the Service Interface	75
9.2	WireFormat	75
9.3	OperationSelector.....	76
9.4	Form of the URI of a Deployed Binding.....	76
9.4.1	Non-hierarchical URIs	76
9.4.2	Determining the URI scheme of a deployed binding.....	76
9.5	SCA Binding	77
9.5.1	Example SCA Binding	78
9.6	Web Service Binding	78
9.7	JMS Binding.....	78
10	SCA Definitions	79
11	Extension Model.....	80
11.1	Defining an Interface Type.....	80
11.2	Defining an Implementation Type.....	82
11.3	Defining a Binding Type.....	83
11.4	Defining an Import Type	85
11.5	Defining an Export Type	87
12	Packaging and Deployment	89
12.1	Domains.....	89
12.2	Contributions.....	89
12.2.1	SCA Artifact Resolution.....	90
12.2.2	SCA Contribution Metadata Document.....	92
12.2.3	Contribution Packaging using ZIP	94
12.3	Installed Contribution	94
12.3.1	Installed Artifact URIs.....	95
12.4	Operations for Contributions.....	95
12.4.1	install Contribution & update Contribution.....	95
12.4.2	add Deployment Composite & update Deployment Composite.....	96
12.4.3	remove Contribution	96
12.5	Use of Existing (non-SCA) Mechanisms for Resolving Artifacts	96

12.6 Domain-Level Composite	96
12.6.1 add To Domain-Level Composite.....	97
12.6.2 remove From Domain-Level Composite	97
12.6.3 get Domain-Level Composite.....	97
12.6.4 get QName Definition	97
12.7 Dynamic Behaviour of Wires in the SCA Domain	98
12.8 Dynamic Behaviour of Component Property Values	98
13 Conformance	99
A. XML Schemas	100
A.1 sca.xsd	100
A.2 sca-core.xsd	100
A.3 sca-binding-sca.xsd.....	108
A.4 sca-interface-java.xsd	109
A.5 sca-interface-wsdl.xsd.....	109
A.6 sca-implementation-java.xsd.....	110
A.7 sca-implementation-composite.xsd.....	111
A.8 sca-definitions.xsd.....	111
A.9 sca-binding-webservice.xsd	112
A.10 sca-binding-jms.xsd.....	112
A.11 sca-policy.xsd.....	112
A.12 sca-contribution.xsd	112
B. SCA Concepts	115
B.1 Binding.....	115
B.2 Component.....	115
B.3 Service.....	115
B.3.1 Remotable Service.....	115
B.3.2 Local Service.....	116
B.4 Reference	116
B.5 Implementation.....	116
B.6 Interface.....	116
B.7 Composite	117
B.8 Composite inclusion	117
B.9 Property	117
B.10 Domain	117
B.11 Wire	117
C. Conformance Items	118
D. Acknowledgements	126
E. Non-Normative Text	127
F. Revision History.....	128

1 Introduction

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

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

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

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

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

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

1.1 Terminology

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

1.2 Normative References

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

[SCA-Java] SCA Java Component Implementation Specification
http://www.osoa.org/download/attachments/35/SCA_JavaComponentImplementation_V100.pdf

[SCA-Common-Java] SCA Java Common Annotations and APIs Specification
http://www.osoa.org/download/attachments/35/SCA_JavaAnnotationsAndAPIs_V100.pdf

[SCA BPEL] SCA BPEL Client and Implementation Specification
<http://docs.oasis-open.org/opencsa/sca-bpel/sca-bpel-1.1-spec-cd-01.pdf>

[SDO] 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

49 [8] WSDL Specification
50 WSDL 1.1: <http://www.w3.org/TR/wsdl>
51 WSDL 2.0: <http://www.w3.org/TR/wsdl20/>
52
53 [9] SCA Web Services Binding Specification
54 <http://docs.oasis-open.org/opencsa/sca-bindings/sca-wsbinding-1.1-spec-cd01.pdf>
55
56 [10] SCA Policy Framework Specification
57 <http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd-01.pdf>
58
59 [11] SCA JMS Binding Specification
60 <http://docs.oasis-open.org/opencsa/sca-bindings/sca-jmsbinding-1.1-spec-cd01.pdf>
61
62 [SCA-CPP-Client] SCA C++ Client and Implementation Specification
63 <http://docs.oasis-open.org/opencsa/sca-c-cpp/sca-cppcni-1.1-spec-cd-01.pdf>
64
65 [SCA-C-Client] SCA C Client and Implementation Specification
66 <http://docs.oasis-open.org/opencsa/sca-c-cpp/sca-ccni-1.1-spec-cd-01.pdf>
67
68 [12] ZIP Format Definition
69 <http://www.pkware.com/documents/casestudies/APPNOTE.TXT>
70
71 [13] Infoset Specification
72 <http://www.w3.org/TR/xml-infoset/>
73
74 [WSDL11_Identifiers] WSDL 1.1 Element Identifiers
75 <http://www.w3.org/TR/wsdl11elementidentifiers/>
76

77 1.3 Naming Conventions

78
79 This specification follows some naming conventions for artifacts defined by the specification,
80 as follows:

- 81
- 82 • For the names of elements and the names of attributes within XSD files, the names follow the
83 CamelCase convention, with all names starting with a lower case letter.
84 eg <element name="componentType" type="sca:ComponentType"/>
 - 85 • For the names of types within XSD files, the names follow the CamelCase convention with all
86 names starting with an upper case letter.
87 eg. <complexType name="ComponentService">
 - 88 • For the names of intents, the names follow the CamelCase convention, with all names starting
89 with a lower case letter, EXCEPT for cases where the intent represents an established acronym,
90 in which case the entire name is in upper case.
91 An example of an intent which is an acronym is the "SOAP" intent.

92 2 Overview

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

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

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

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

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

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

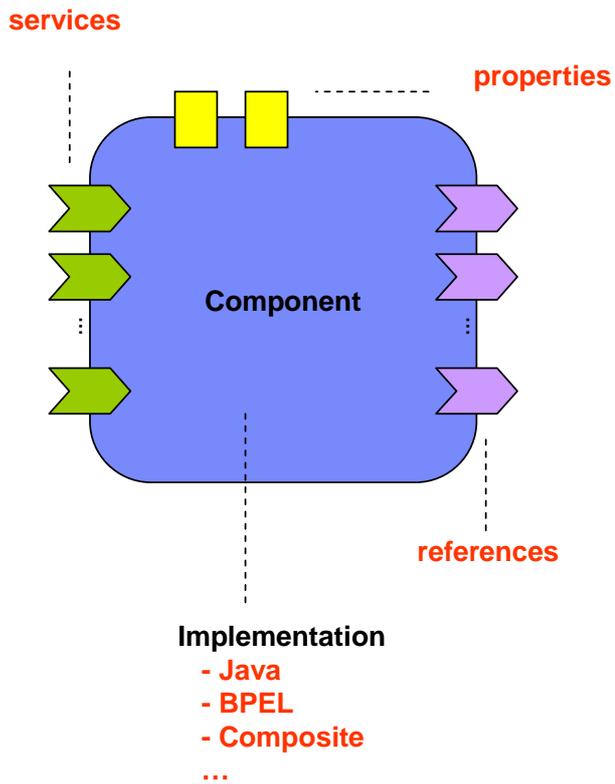
130 Composites are deployed within an **SCA Domain**. An SCA Domain typically represents a set of services
131 providing an area of business functionality that is controlled by a single organization. As an example, for
132 the accounts department in a business, the SCA Domain might cover all financial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

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

142 **2.1 Diagram used to Represent SCA Artifacts**

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

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



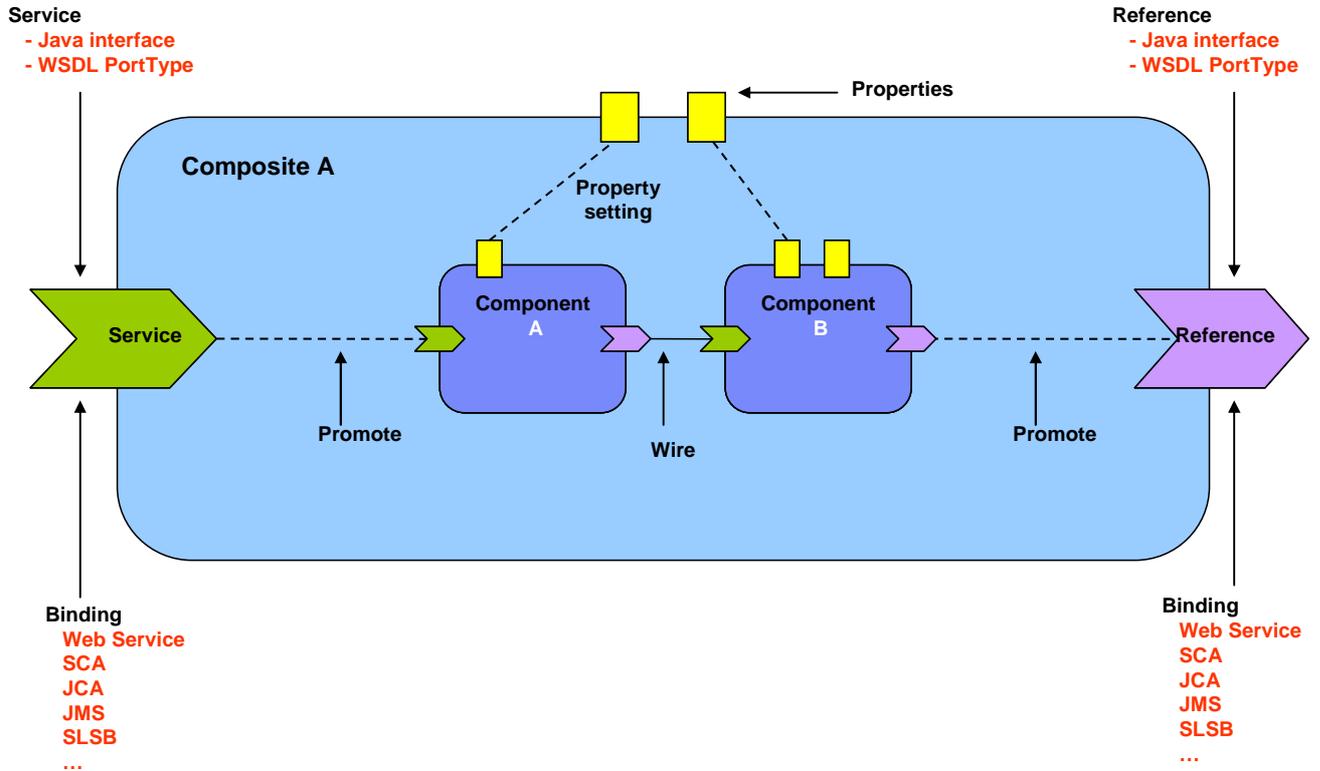
147

148 *Figure 1: SCA Component Diagram*

149

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

152

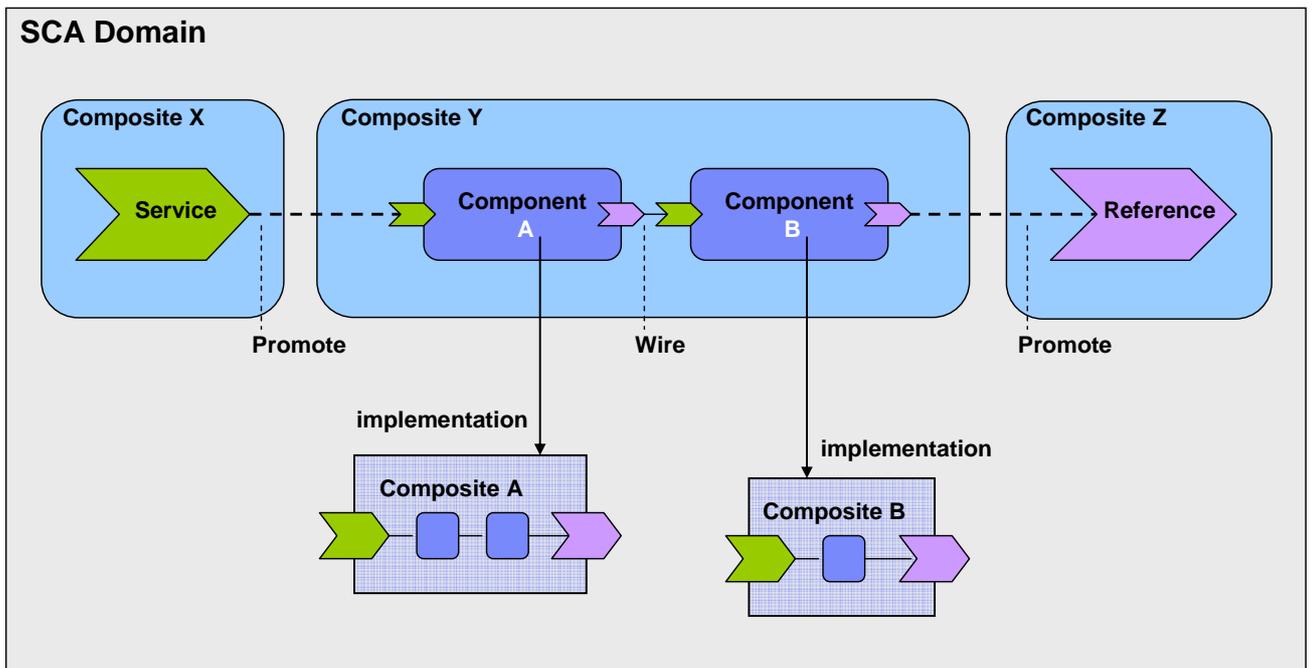


153

154 *Figure 2: SCA Composite Diagram*

155

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



158

159 *Figure 3: SCA Domain Diagram*

160 **3 Quick Tour by Sample**

161 To be completed.

162

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

164

165

4 Implementation and ComponentType

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

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

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

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

So, for example:

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

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

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

4.1 Component Type

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

An implementation type specification (for example, the WS-BPEL Client and Implementation Specification Version 1.1 [SCA BPEL]) specifies the mechanism(s) by which the component type associated with an implementation of that type is derived.

Since SCA allows a broad range of implementation technologies, it is expected that some implementation technologies (for example, the Java Component Implementation Specification Version 1.1 [SCA-Java]) allow for introspecting the implementation artifact(s) (for example, a Java class) to derive the component type information. Other implementation technologies might not allow for introspection of the implementation artifact(s). In those cases where introspection is not allowed, SCA encourages the use of a SCA component type side file. A **component type side file** is an XML file whose document root element is `sca:componentType`.

211 The implementation type specification defines whether introspection is allowed, whether a side file
212 is allowed, both are allowed or some other mechanism specifies the component type. The
213 component type information derived through introspection is called the **introspected component**
214 **type**. In any case, the implementation type specification specifies how multiple sources of
215 information are combined to produce the **effective component type**. The effective component
216 type is the component type metadata that is presented to the using Component for configuration.

217 The extension of a componentType side file name MUST be .componentType. [ASM40001] The
218 name and location of a componentType side file, if allowed, is defined by the implementation type
219 specification.

220 If a component type side file is not allowed for a particular implementation type, the effective
221 component type and introspected component type are one and the same for that implementation
222 type.

223 For the rest of this document, when the term 'component type' is used it refers to the 'effective
224 component type'.

225 The following snippet shows the componentType pseudo-schema:

226

```
227 <?xml version="1.0" encoding="ASCII"?>  
228 <!-- Component type schema snippet -->  
229 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
230     constrainingType="QName"? >  
231  
232     <service ... />*  
233     <reference ... />*  
234     <property ... />*  
235     <implementation ... />?  
236  
237 </componentType>  
238
```

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

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

245

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

- 247 • **service : Service (0..n)** – see [component type service section](#).
- 248 • **reference : Reference (0..n)** – see [component type reference section](#).
- 249 • **property : Property (0..n)** – see [component type property section](#).
- 250 • **implementation : Implementation (0..1)** – see [component type implementation](#)
251 [section](#).

252

253 4.1.1 Service

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

258

```

259 <?xml version="1.0" encoding="ASCII"?>
260 <!-- Component type service schema snippet -->
261 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ...
262 >
263
264     <service name="xs:NCName"
265         requires="list of xs:QName"? policySets="list of xs:QName"?>*
266         <interface ... />*
267         <operation name="xs:NCName" requires="list of xs:QName"?
268             policySets="list of xs:QName"?/>*
269         <binding ... />*
270         <callback?
271             <binding ... />+
272         </callback>
273     </service>
274
275     <reference ... />*
276     <property ... />*
277     <implementation ... />?
278
279 </componentType>
280

```

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

- 282 • **name : NCName (1..1)** - the name of the service. The @name attribute of a <service/>
- 283 child element of a <componentType/> MUST be unique amongst the service elements of
- 284 that <componentType/>. [ASM40003]
- 285 • **requires : QName (0..n)** - a list of policy intents. See the [Policy Framework specification](#)
- 286 [10] for a description of this attribute.
- 287 • **policySets : QName (0..n)** - a list of policy sets. See the [Policy Framework specification](#)
- 288 [10] for a description of this attribute.

289

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

- 291 • **interface : Interface (1..1)** - A service has **one interface**, which describes the
- 292 operations provided by the service. For details on the interface element see [the Interface](#)
- 293 [section](#).
- 294 • **operation: Operation (0..n)** - Zero or more operation elements. These elements are
- 295 used to describe characteristics of individual operations within the interface. For a detailed
- 296 description of the operation element, see [the Policy Framework specification](#) [SCA Policy].
- 297 • **binding : Binding (0..n)** - A service element has **zero or more binding elements** as
- 298 children. If the binding element is not present it defaults to <binding.sca>. Details of the
- 299 binding element are described in [the Bindings section](#).
- 300 • **callback (0..1) / binding : Binding (1..n)** - A service element has an optional **callback**
- 301 element used if the interface has a callback defined, which has one or more **binding**
- 302 elements as children. The **callback** and its binding child elements are specified if there is
- 303 a need to have binding details used to handle callbacks. If the callback element is not
- 304 present, the behaviour is runtime implementation dependent. For details on callbacks, see
- 305 [the Bidirectional Interfaces section](#).

306

307 4.1.2 Reference

308 A **Reference** represents a requirement that the implementation has on a service provided by

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

310 componentType element. There can be **zero or more** reference elements in a component type
311 definition. The following snippet shows the component type schema with the schema for a
312 reference child element:

313

```
314 <?xml version="1.0" encoding="ASCII"?>
315 <!-- Component type reference schema snippet -->
316 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ...
317 >
318
319     <service ... />*
320
321     <reference name="xs:NCName"
322               autowire="xs:boolean"?
323               multiplicity="0..1 or 1..1 or 0..n or 1..n"?
324               wiredByImpl="xs:boolean"?
325               requires="list of xs:QName"? policySets="list of xs:QName"?>*
326     <interface ... />
327     <operation name="xs:NCName" requires="list of xs:QName"?
328               policySets="list of xs:QName"?/>*
329     <binding ... />*
330     <callback?
331         <binding ... />+
332     </callback>
333 </reference>
334
335 <property ... />*
336 <implementation ... />?
337
338 </componentType>
339
```

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

- 341 • **name : NCName (1..1)** - the name of the reference. The @name attribute of a
342 <reference/> child element of a <componentType/> MUST be unique amongst the
343 reference elements of that <componentType/>. [ASM40004]
- 344 • **multiplicity : 0..1|1..1|0..n|1..n (0..1)** - defines the number of wires that can connect
345 the reference to target services. The multiplicity can have the following values
 - 346 ○ 0..1 – zero or one wire can have the reference as a source
 - 347 ○ 1..1 – one wire can have the reference as a source
 - 348 ○ 0..n - zero or more wires can have the reference as a source
 - 349 ○ 1..n – one or more wires can have the reference as a source

350 If @multiplicity is not specified, the default value is "1..1".

- 351 • **autowire : boolean (0..1)** - whether the reference should be autowired, as described in
352 the [Autowire section](#). Default is false.
- 353 • **wiredByImpl : boolean (0..1)** - a boolean value, "false" by default. If set to "false", the
354 reference is wired to the target(s) configured on the reference. If set to "true" it indicates
355 that the target of the reference is set at runtime by the implementation code (eg by the
356 code obtaining an endpoint reference by some means and setting this as the target of the
357 reference through the use of programming interfaces defined by the relevant Client and
358 Implementation specification). If @wiredByImpl is set to "true", then any reference
359 targets configured for this reference MUST be ignored by the runtime. [ASM40006] It is
360 recommended that any references with @wiredByImpl = "true" are left unwired.

- 361 • **requires : QName (0..n)** - a list of policy intents. See the [Policy Framework specification](#)
362 [\[10\]](#) for a description of this attribute.
- 363 • **policySets : QName (0..n)** - a list of policy sets. See the [Policy Framework specification](#)
364 [\[10\]](#) for a description of this attribute.

365

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

- 367 • **interface : Interface (1..1)** - A reference has **one interface**, which describes the
368 operations required by the reference. The interface is described by an **interface element**
369 which is a child element of the reference element. For details on the interface element see
370 [the Interface section](#).
- 371 • **operation: Operation (0..n)** - Zero or more operation elements. These elements are
372 used to describe characteristics of individual operations within the interface. For a detailed
373 description of the operation element, see [the Policy Framework specification \[SCA Policy\]](#).
- 374 • **binding : Binding (0..n)** - A reference element has **zero or more binding elements** as
375 children. Details of the binding element are described in the [Bindings section](#).

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

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

390

391 4.1.3 Property

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

396

```
397 <?xml version="1.0" encoding="ASCII"?>
398 <!-- Component type property schema snippet -->
399 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ...
400 >
401
402 <service ... /*
403 <reference ... /*
404
405 <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
406     many="xs:boolean"? mustSupply="xs:boolean"?
407     requires="list of xs:QName"?
408     policySets="list of xs:QName"?>*
409     default-property-value?
410 </property>
411
```

```
412     <implementation ... />?
413
414 </componentType>
415
```

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

- 417 ▪ **name : NCName (1..1)** - the name of the property. The @name attribute of a
418 <property/> child element of a <componentType/> MUST be unique amongst the
419 property elements of that <componentType/>. [ASM40005]
- 420 ▪ one of **(1..1)**:
 - 421 ○ **type : QName** - the type of the property defined as the qualified name of an XML
422 schema type. The value of the property @type attribute MUST be the QName of
423 an XML schema type. [ASM40007]
 - 424 ○ **element : QName** - the type of the property defined as the qualified name of an
425 XML schema global element – the type is the type of the global element. The value
426 of the property @element attribute MUST be the QName of an XSD global
427 element. [ASM40008]
- 428 ▪ **many : boolean (0..1)** - (optional) whether the property is single-valued (false) or multi-
429 valued (true). In the case of a multi-valued property, it is presented to the implementation
430 as a collection of property values. If many is not specified, it takes a default value of false.
- 431 ▪ **mustSupply : boolean (0..1)** - whether the property value must be supplied by the
432 component that uses the implementation – when mustSupply="true" the component must
433 supply a value since the implementation has no default value for the property. A default-
434 property-value should only be supplied when mustSupply="false" (the default setting for
435 the mustSupply attribute), since the implication of a default value is that it is used only
436 when a value is not supplied by the using component. If mustSupply is not specified, it
437 takes a default value of false.
- 438 ▪ **file : anyURI (0..1)** - a dereferencable URI to a file containing a value for the property.
- 439 ▪ **requires : QName (0..n)** - a list of policy intents. See the [Policy Framework specification](#)
440 [\[10\]](#) for a description of this attribute.
- 441 ▪ **policySets : QName (0..n)** - a list of policy sets. See the [Policy Framework specification](#)
442 [\[10\]](#) for a description of this attribute.

443 The property element can contain a default property value as its content. The form of the default
444 property value is as described [in the section on Component Property](#).

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

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

458

459 4.1.4 Implementation

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

462 policies. The following snippet shows the component type schema with the schema for a
463 implementation child element:

```
464  
465 <?xml version="1.0" encoding="ASCII"?>  
466 <!-- Component type implementation schema snippet -->  
467 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ...  
468 >  
469  
470     <service ... /> *  
471     <reference ... > *  
472     <property ... /> *  
473  
474     <implementation requires="list of xs:QName"?  
475                 policySets="list of xs:QName"? /> ?  
476  
477 </componentType>  
478
```

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

- 480 • **requires : QName (0..n)** - a list of policy intents. See the [Policy Framework specification](#)
481 [\[10\]](#) for a description of this attribute.
- 482 • **policySets : QName (0..n)** - a list of policy sets. See the [Policy Framework specification](#)
483 [\[10\]](#) for a description of this attribute.

484

485 4.2 Example ComponentType

486

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

```
490  
491 <?xml version="1.0" encoding="ASCII"?>  
492 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">  
493  
494     <service name="MyValueService">  
495         <interface.java interface="services.myvalue.MyValueService"/>  
496     </service>  
497  
498     <reference name="customerService">  
499         <interface.java interface="services.customer.CustomerService"/>  
500     </reference>  
501     <reference name="stockQuoteService">  
502         <interface.java  
503             interface="services.stockquote.StockQuoteService"/>  
504     </reference>  
505  
506     <property name="currency" type="xsd:string">USD</property>  
507  
508 </componentType>  
509
```

510 4.3 Example Implementation

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

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

```
515  
516 package services.account;  
517  
518 @Remotable  
519 public interface AccountService {  
520  
521     AccountReport getAccountReport(String customerID);  
522 }  
523
```

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

```
527  
528 package services.account;  
529  
530 import java.util.List;  
531  
532 import commonj.sdo.DataFactory;  
533  
534 import org.oesa.sca.annotations.Property;  
535 import org.oesa.sca.annotations.Reference;  
536  
537 import services.accountdata.AccountDataService;  
538 import services.accountdata.CheckingAccount;  
539 import services.accountdata.SavingsAccount;  
540 import services.accountdata.StockAccount;  
541 import services.stockquote.StockQuoteService;  
542  
543 public class AccountServiceImpl implements AccountService {  
544  
545     @Property  
546     private String currency = "USD";  
547  
548     @Reference  
549     private AccountDataService accountDataService;  
550     @Reference  
551     private StockQuoteService stockQuoteService;  
552  
553     public AccountReport getAccountReport(String customerID) {  
554  
555         DataFactory dataFactory = DataFactory.INSTANCE;  
556         AccountReport accountReport = (AccountReport) dataFactory.create(AccountReport.class);  
557         List accountSummaries = accountReport.getAccountSummaries();  
558  
559         CheckingAccount checkingAccount = accountDataService.getCheckingAccount(customerID);  
560         AccountSummary checkingAccountSummary =  
561             (AccountSummary) dataFactory.create(AccountSummary.class);  
562         checkingAccountSummary.setAccountNumber(checkingAccount.getAccountNumber());  
563         checkingAccountSummary.setAccountType("checking");  
564         checkingAccountSummary.setBalance(fromUSDollarToCurrency(checkingAccount.getBalance()));  
565         accountSummaries.add(checkingAccountSummary);  
566  
567         SavingsAccount savingsAccount = accountDataService.getSavingsAccount(customerID);  
568         AccountSummary savingsAccountSummary =  
569             (AccountSummary) dataFactory.create(AccountSummary.class);  
570         savingsAccountSummary.setAccountNumber(savingsAccount.getAccountNumber());  
571         savingsAccountSummary.setAccountType("savings");  
572         savingsAccountSummary.setBalance(fromUSDollarToCurrency(savingsAccount.getBalance()));  
573         accountSummaries.add(savingsAccountSummary);  
574
```

```

575     StockAccount stockAccount = accountDataService.getStockAccount(customerID);
576     AccountSummary stockAccountSummary =
577         (AccountSummary)dataFactory.create(AccountSummary.class);
578     stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());
579     stockAccountSummary.setAccountType("stock");
580     float balance=
581         (stockQuoteService.getQuote(stockAccount.getSymbol()))*stockAccount.getQuantity();
582     stockAccountSummary.setBalance(fromUSDollarToCurrency(balance));
583     accountSummaries.add(stockAccountSummary);
584
585     return accountReport;
586 }
587
588 private float fromUSDollarToCurrency(float value){
589
590     if (currency.equals("USD")) return value; else
591     if (currency.equals("EURO")) return value * 0.8f; else
592     return 0.0f;
593 }
594 }
595

```

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

```

598
599 <?xml version="1.0" encoding="ASCII"?>
600 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
601               xmlns:xsd="http://www.w3.org/2001/XMLSchema">
602
603     <service name="AccountService">
604         <interface.java interface="services.account.AccountService"/>
605     </service>
606     <reference name="accountDataService">
607         <interface.java
608             interface="services.accountdata.AccountDataService"/>
609     </reference>
610     <reference name="stockQuoteService">
611         <interface.java
612             interface="services.stockquote.StockQuoteService"/>
613     </reference>
614
615     <property name="currency" type="xsd:string">USD</property>
616
617 </componentType>
618

```

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

5 Component

622

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

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

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

632

```
633 <?xml version="1.0" encoding="UTF-8"?>
634 <!-- Component schema snippet -->
635 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
636   ...
637   <component name="xs:NCName" autowire="xs:boolean"?
638             requires="list of xs:QName"? policySets="list of xs:QName"?
639             constrainingType="xs:QName"?>*
640     <implementation ... />?
641     <service ... />*
642     <reference ... />*
643     <property ... />*
644   </component>
645   ...
646 </composite>
647
```

647

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

- 649 • **name : NCName (1..1)** – the name of the component. The @name attribute of a
650 <component/> child element of a <composite/> MUST be unique amongst the component
651 elements of that <composite/> [ASM50001]
- 652 • **autowire : boolean (0..1)** – whether contained component references should be
653 autowired, as described in [the Autowire section](#). Default is false.
- 654 • **requires : QName (0..n)** – a list of policy intents. See the [Policy Framework specification](#)
655 [10] for a description of this attribute.
- 656 • **policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification](#)
657 [10] for a description of this attribute.
- 658 • **constrainingType : QName (0..1)** – the name of a constrainingType. When specified,
659 the set of services, references and properties of the component, plus related intents, is
660 constrained to the set defined by the constrainingType. See [the ConstrainingType Section](#)
661 for more details.

662

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

- 664 • **implementation : ComponentImplementation (0..1)** – see [component](#)
665 [implementation section](#).
- 666 • **service : ComponentService (0..n)** – see [component service section](#).
- 667 • **reference : ComponentReference (0..n)** – see [component reference section](#).
- 668 • **property : ComponentProperty (0..n)** – see [component property section](#).

669

670 5.1 Implementation

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

676

```
677 <?xml version="1.0" encoding="UTF-8"?>
678 <!-- Component Implementation schema snippet -->
679 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
680   ...
681   <component ... >*
682     <implementation ... />?
683     <service ... />*
684     <reference ... />*
685     <property ... />*
686   </component>
687   ...
688 </composite>
689
```

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

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

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

700

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

702

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

704

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

706

707

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

709

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

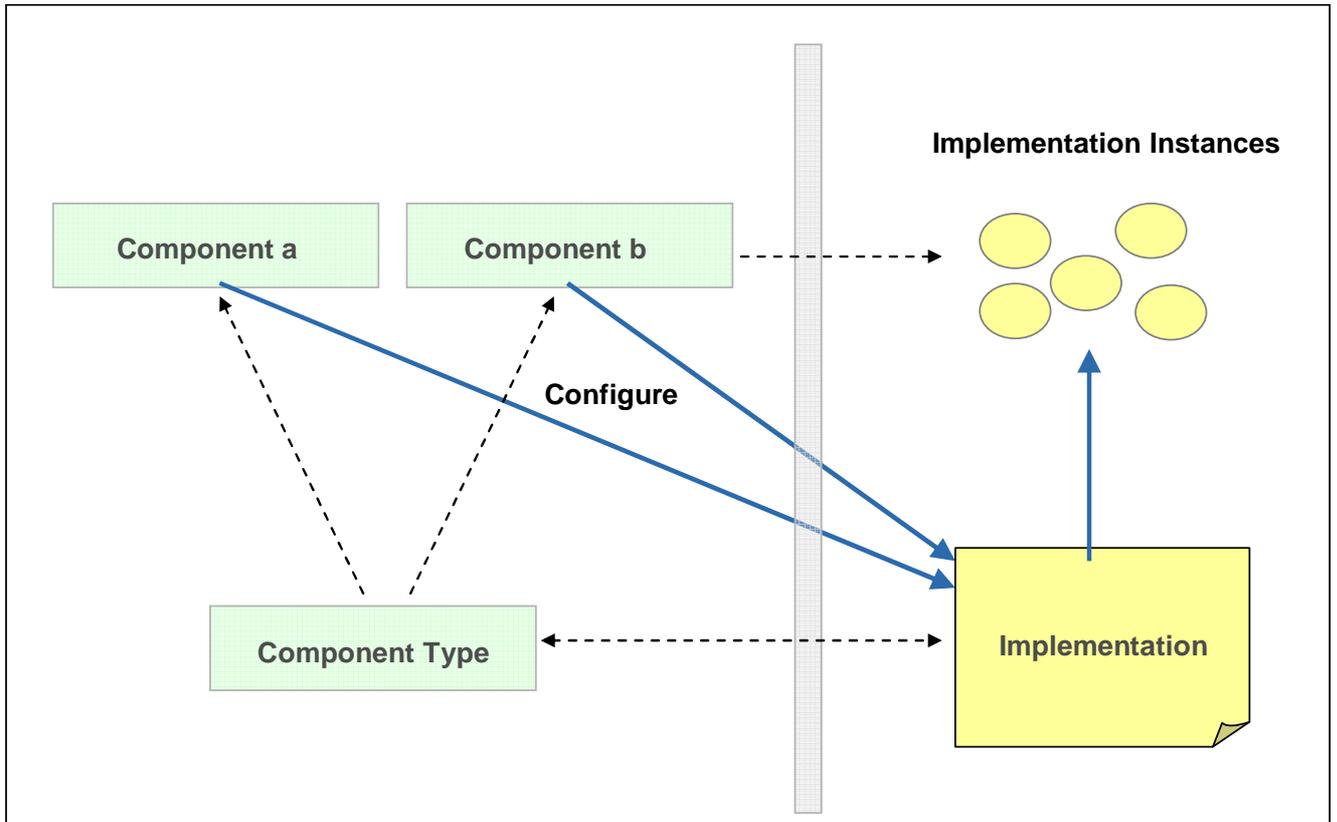


Figure 4: Relationship of Component and Implementation

5.2 Service

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

```

724 <?xml version="1.0" encoding="UTF-8"?>
725 <!-- Component Service schema snippet -->
726 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
727   ...
728   <component ... >*
729     <implementation ... />?
730     <service name="xs:NCName" requires="list of xs:QName"?
731       policySets="list of xs:QName"?>*
732       <interface ... />?
733       <operation name="xs:NCName" requires="list of xs:QName"?
734         policySets="list of xs:QName"?/>*
735       <binding ... />*
736       <callback?
737         <binding ... />+
738       </callback>
739     </service>
740     <reference ... />*
741     <property ... />*
742   </component>

```

743 ...
744 </composite>
745

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

- 747 • **name : NCName (1..1)** - the name of the service. The @name attribute of a service
748 element of a <component/> MUST be unique amongst the service elements of that
749 <component/> [ASM50002] The @name attribute of a service element of a
750 <component/> MUST match the @name attribute of a service element of the
751 componentType of the <implementation/> child element of the component. [ASM50003]
- 752 • **requires : QName (0..n)** – a list of policy intents. See the [Policy Framework specification](#)
753 [10] for a description of this attribute.
754 Note: The effective set of policy intents for the service consists of any intents explicitly
755 stated in this requires attribute, combined with any intents specified for the service by the
756 implementation.
- 757 • **policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification](#)
758 [10] for a description of this attribute.

759

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

- 761 • **interface : Interface (0..1)** - A service has **zero or one interface**, which describes the
762 operations provided by the service. The interface is described by an **interface element**
763 which is a child element of the service element. If no interface is specified, then the
764 interface specified for the service in the componentType of the implementation is in effect.
765 If a <service/> element has an interface subelement specified, the interface MUST provide
766 a compatible subset of the interface declared on the componentType of the
767 implementation [ASM50004] For details on the interface element see [the Interface section](#).
- 768 • **operation: Operation (0..n)** - Zero or more operation elements. These elements are
769 used to describe characteristics of individual operations within the interface. For a detailed
770 description of the operation element, see [the Policy Framework specification](#) [SCA Policy].
- 771 • **binding : Binding (0..n)** - A service element has **zero or more binding elements** as
772 children. If no binding elements are specified for the service, then the bindings specified
773 for the equivalent service in the componentType of the implementation MUST be used, but
774 if the componentType also has no bindings specified, then <binding.sca/> MUST be used
775 as the binding. If binding elements are specified for the service, then those bindings MUST
776 be used and they override any bindings specified for the equivalent service in the
777 componentType of the implementation. [ASM50005] Details of the binding element are
778 described in [the Bindings section](#). The binding, combined with any PolicySets in effect for
779 the binding, needs to satisfy the set of policy intents for the service, as described in [the](#)
780 [Policy Framework specification](#) [10].
- 781 • **callback (0..1) / binding : Binding (1..n)** - A service element has an optional **callback**
782 element used if the interface has a callback defined, which has one or more **binding**
783 elements as children. The **callback** and its binding child elements are specified if there is
784 a need to have binding details used to handle callbacks. If the callback element is present
785 and contains one or more binding child elements, then those bindings MUST be used for
786 the callback. [ASM50006] If the callback element is not present, the behaviour is runtime
787 implementation dependent.

788

789 5.3 Reference

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

```

794
795 <?xml version="1.0" encoding="UTF-8"?>
796 <!-- Component Reference schema snippet -->
797 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
798   ...
799   <component ... >*
800     <implementation ... />?
801     <service ... />*
802     <reference name="xs:NCName"
803       target="list of xs:anyURI"? autowire="xs:boolean"?
804       multiplicity="0..1 or 1..1 or 0..n or 1..n"?
805       wiredByImpl="xs:boolean"? requires="list of xs:QName"?
806       policySets="list of xs:QName"?>*
807     <interface ... />?
808     <operation name="xs:NCName" requires="list of xs:QName"?
809       policySets="list of xs:QName"?/>*
810     <binding uri="xs:anyURI"? requires="list of xs:QName"?
811       policySets="list of xs:QName"?/>*
812     <callback?
813       <binding ... />+
814     </callback>
815   </reference>
816   <property ... />*
817 </component>
818   ...
819 </composite>
820

```

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

- 822 • **name : NCName (1..1)** – the name of the reference. The @name attribute of a service
823 element of a <component/> MUST be unique amongst the service elements of that
824 <component/> [ASM50007] The @name attribute of a reference element of a
825 <component/> MUST match the @name attribute of a reference element of the
826 componentType of the <implementation/> child element of the component. [ASM50008]
- 827 • **autowire : boolean (0..1)** – whether the reference should be autowired, as described in
828 the [Autowire section](#). Default is false.
- 829 • **requires : QName (0..n)** – a list of policy intents. See the [Policy Framework specification](#)
830 [10] for a description of this attribute.
831 Note: The effective set of policy intents for the reference consists of any intents explicitly
832 stated in this requires attribute, combined with any intents specified for the reference by
833 the implementation.
- 834 • **policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification](#)
835 [10] for a description of this attribute.
- 836 • **multiplicity : 0..1|1..1|0..n|1..n (0..1)** - defines the number of wires that can connect
837 the reference to target services. Overrides the multiplicity specified for this reference in
838 the componentType of the implementation. The multiplicity can have the following values
 - 839 ○ 0..1 – zero or one wire can have the reference as a source
 - 840 ○ 1..1 – one wire can have the reference as a source
 - 841 ○ 0..n - zero or more wires can have the reference as a source
 - 842 ○ 1..n – one or more wires can have the reference as a source

843 The value of multiplicity for a component reference MUST only be equal or further restrict
844 any value for the multiplicity of the reference with the same name in the componentType
845 of the implementation, where further restriction means 0..n to 0..1 or 1..n to 1..1.
846 [ASM50009]

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

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

861

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

- 863 • **interface : Interface (0..1)** - A reference has **zero or one interface**, which describes
864 the operations required by the reference. The interface is described by an **interface**
865 **element** which is a child element of the reference element. If no interface is specified,
866 then the interface specified for the reference in the `componentType` of the implementation
867 is in effect. If an interface is declared for a component reference it MUST provide a
868 compatible superset of the interface declared for the equivalent reference in the
869 `componentType` of the implementation, i.e. provide the same operations or a superset of
870 the operations defined by the implementation for the reference. [\[ASM50011\]](#) For details
871 on the interface element see [the Interface section](#).

- 872 • **operation: Operation (0..n)** - Zero or more operation elements. These elements are
873 used to describe characteristics of individual operations within the interface. For a detailed
874 description of the operation element, see [the Policy Framework specification](#) [SCA Policy].

- 875 • **binding : Binding (0..n)** - A reference element has **zero or more binding elements** as
876 children. If no binding elements are specified for the reference, then the bindings specified
877 for the equivalent reference in the `componentType` of the implementation MUST be used,
878 but if the `componentType` also has no bindings specified, then `<binding.sca/>` MUST be
879 used as the binding. If binding elements are specified for the reference, then those
880 bindings MUST be used and they override any bindings specified for the equivalent
881 reference in the `componentType` of the implementation. [\[ASM50012\]](#) Details of the binding
882 element are described in the [Bindings section](#). The binding, combined with any `PolicySets`
883 in effect for the binding, needs to satisfy the set of policy intents for the reference, as
884 described in [the Policy Framework specification](#) [10].

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

- 888 • **callback (0..1) / binding : Binding (1..n)** - A **reference** element has an optional
889 **callback** element used if the interface has a callback defined, which has one or more
890 **binding** elements as children. The **callback** and its binding child elements are specified if
891 there is a need to have binding details used to handle callbacks. If the callback element is
892 present and contains one or more binding child elements, then those bindings MUST be
893 used for the callback. [\[ASM50006\]](#) If the callback element is not present, the behaviour is
894 runtime implementation dependent.

895 5.3.1 Specifying the Target Service(s) for a Reference

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

- 898 1. Through a value specified in the `@target` attribute of the reference element

- 899 2. Through a target URI specified in the @uri attribute of a binding element which is a child
900 of the reference element
- 901 3. Through the setting of one or more values for binding-specific attributes and/or child
902 elements of a binding element that is a child of the reference element
- 903 4. Through the specification of @autowire="true" for the reference (or through inheritance
904 of that value from the component or composite containing the reference)
- 905 5. Through the specification of @wiredByImpl="true" for the reference
- 906 6. Through the promotion of a component reference by a composite reference of the
907 composite containing the component (the target service is then identified by the
908 configuration of the composite reference)
- 909 7. Through the presence of a <wire/> element which has the reference specified in its
910 @source attribute.
- 911 Combinations of these different methods are allowed, and the following rules MUST be observed:
- 912 • If @wiredByImpl="true", other methods of specifying the target service MUST NOT be
913 used. [ASM50013]
 - 914 • If @autowire="true", the autowire procedure MUST only be used if no target is identified
915 by any of the other ways listed above. It is not an error if @autowire="true" and a target
916 is also defined through some other means, however in this case the autowire procedure
917 MUST NOT be used. [ASM50014]
 - 918 • If a reference has a value specified for one or more target services in its @target attribute,
919 there MUST NOT be any child <binding/> elements declared for that reference.
920 [ASM50026]
 - 921 • If a binding element has a value specified for a target service using its @uri attribute, the
922 binding element MUST NOT identify target services using binding specific attributes or
923 elements. [ASM50015]
 - 924 • It is possible that a particular binding type MAY require that the address of a target service
925 uses more than a simple URI. In such cases, the @uri attribute MUST NOT be used to
926 identify the target service - instead, binding specific attributes and/or child elements must
927 be used. [ASM50016]
 - 928 • If any <wire/> element with its @replace attribute set to "true" has a particular reference
929 specified in its @source attribute, the value of the @target attribute for that reference
930 MUST be ignored and MUST NOT be used to define target services for that reference.
931 [ASM50034]

932 5.3.1.1 Multiplicity and the Valid Number of Target Services for a Reference

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

- 934 • A reference with multiplicity 0..1 or 0..n MAY have no target service defined. [ASM50018]
- 935 • A reference with multiplicity 0..1 or 1..1 MUST NOT have more than one target service
936 defined. [ASM50019]
- 937 • A reference with multiplicity 1..1 or 1..n MUST have at least one target service defined.
938 [ASM50020]
- 939 • A reference with multiplicity 0..n or 1..n MAY have one or more target services defined.
940 [ASM50021]

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

944 Some reference multiplicity errors can be detected at deployment time. In these cases, an error
945 SHOULD be generated by the SCA runtime at deployment time. [ASM50023] For example, where
946 a composite is used as a component implementation, wires and target services cannot be added to

947 the composite after deployment. As a result, for components which are part of the composite,
948 both missing wires and wires with a non-existent target can be detected at deployment time
949 through a scan of the contents of the composite.

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

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

963 5.4 Property

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

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

- 971 • As a value, supplied in the **value** attribute of the property element.
972 If the @value attribute of a component property element is declared, the type of the
973 property MUST be an XML Schema simple type and the @value attribute MUST contain a
974 single value of that type. [ASM50027]

975 For example,

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

- 977 • As a value, supplied as the content of the **value** element(s) children of the property
978 element.
979 If the value subelement of a component property is specified, the type of the property
980 MUST be an XML Schema simple type or an XML schema complex type. [ASM50028]

981 For example,

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

```
984 <property name="pi">  
985   <value>3.14159265</value>  
986 </property>
```

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

```
989 <property name="currency">  
990   <value>EURO</value>  
991   <value>USDollar</value>  
992 </property>
```

- 993 • property defined using a XML Schema complex type and which contains a single
994 value

```
995 <property name="complexFoo">  
996   <value attr="bar">  
997     <foo:a>TheValue</foo:a>  
998     <foo:b>InterestingURI</foo:b>
```

- 999 </value>
 1000 </property>
- 1001 • property defined using a XML Schema complex type and which contains multiple
 1002 values
- ```

1003 <property name="complexBar">
1004 <value anotherAttr="foo">
1005 <bar:a>AValue</bar:a>
1006 <bar:b>InterestingURI</bar:b>
1007 </value>
1008 <value attr="zing">
1009 <bar:a>BValue</bar:a>
1010 <bar:b>BoringURI</bar:b>
1011 </value>
1012 </property>

```
- 1013                  • As a value, supplied as the content of the property element.  
 1014                  If a component property value is declared using a child element of the <property/>  
 1015                  element, the type of the property MUST be an XML Schema global element and the  
 1016                  declared child element MUST be an instance of that global element. [\[ASM50029\]](#)

1017                  For example,

- 1018                  • property defined using a XML Schema global element declaration and which  
 1019                  contains a single value

```

1020 <property name="foo">
1021 <foo:SomeGED ...>...</foo:SomeGED>
1022 </property>

```

- 1023                  • property defined using a XML Schema global element declaration and which  
 1024                  contains multiple values

```

1025 <property name="bar">
1026 <bar:SomeOtherGED ...>...</bar:SomeOtherGED>
1027 <bar:SomeOtherGED ...>...</bar:SomeOtherGED>
1028 </property>

```

- 1029                  • By referencing a Property value of the composite which contains the component. The  
 1030                  reference is made using the **source** attribute of the property element.

1031  
 1032                  The form of the value of the source attribute follows the form of an XPath expression.  
 1033                  This form allows a specific property of the composite to be addressed by name. Where the  
 1034                  composite property is of a complex type, the XPath expression can be extended to refer to  
 1035                  a sub-part of the complex property value.

1036  
 1037                  So, for example, `source="$currency"` is used to reference a property of the composite  
 1038                  called "currency", while `source="$currency/a"` references the sub-part "a" of the  
 1039                  complex composite property with the name "currency".

- 1040                  • By specifying a dereferencable URI to a file containing the property value through the **file**  
 1041                  attribute. The contents of the referenced file are used as the value of the property.

1042  
 1043                  If more than one property value specification is present, the source attribute takes precedence, then  
 1044                  the file attribute.

1045                  For a property defined using a XML Schema simple type and for which a single value is desired, can  
 1046                  be set either using the @value attribute or the <value> child element. The two forms in such a case  
 1047                  are equivalent.

1048                  When a property has multiple values set, they MUST all be contained within the same property  
 1049                  element. A <component/> element MUST NOT contain two <property/> subelements with the same  
 1050                  value of the @name attribute. [\[ASM50030\]](#)

1051                  Optionally, the type of the property can be specified in **one** of two ways:

- 1052                  • by the qualified name of a type defined in an XML schema, using the **type** attribute

- by the qualified name of a global element in an XML schema, using the `element` attribute
- The property type specified must be compatible with the type of the property declared in the component type of the implementation. If no type is declared in the component property, the type of the property declared by the implementation is used.

1057

1058 The following snippet shows the component schema with the schema for a property child element:

1059

```

1060 <?xml version="1.0" encoding="UTF-8"?>
1061 <!-- Component Property schema snippet -->
1062 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
1063 ...
1064 <component ... >*
1065 <implementation ... />?
1066 <service ... />*
1067 <reference ... />*
1068 <property name="xs:NCName"
1069 (type="xs:QName" | element="xs:QName")?
1070 mustSupply="xs:boolean"? many="xs:boolean"?
1071 source="xs:string"? file="xs:anyURI"?
1072 requires="list of xs:QName"?
1073 policySets="list of xs:QName"?
1074 value="xs:string"?>*
1075 [<value>+ | xs:any+]?
1076 </property>
1077 </component>
1078 ...
1079 </composite>

```

1080

1081 The **component property** element has the following **attributes**:

1082

- **name : NCName (1..1)** – the name of the property. The name attribute of a component property MUST match the name of a property element in the component type of the component implementation. [\[ASM50031\]](#)

1085

- zero or one of **(0..1)**:
  - **type : QName** – the type of the property defined as the qualified name of an XML schema type

1088

- **element : QName** – the type of the property defined as the qualified name of an XML schema global element – the type is the type of the global element

1090

- **source : string (0..1)** – an XPath expression pointing to a property of the containing composite from which the value of this component property is obtained.

1092

- **file : anyURI (0..1)** – a dereferencable URI to a file containing a value for the property

1093

- **many : boolean (0..1)** – (optional) whether the property is single-valued (false) or multi-valued (true). Overrides the many specified for this property on the implementation. The value can only be equal or further restrict, i.e. if the implementation specifies many true, then the component can say false. In the case of a multi-valued property, it is presented to the implementation as a Collection of property values. If many is not specified, it takes the value defined by the component type of the implementation used by the component.

1100

- **value : string (0..1)** – the value of the property if the property is defined using a simple type.

1102

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

1103

1104       ▪ **policySets : QName (0..n)** - a list of policy sets. See the [Policy Framework specification](#)  
1105       [\[10\]](#) for a description of this attribute.

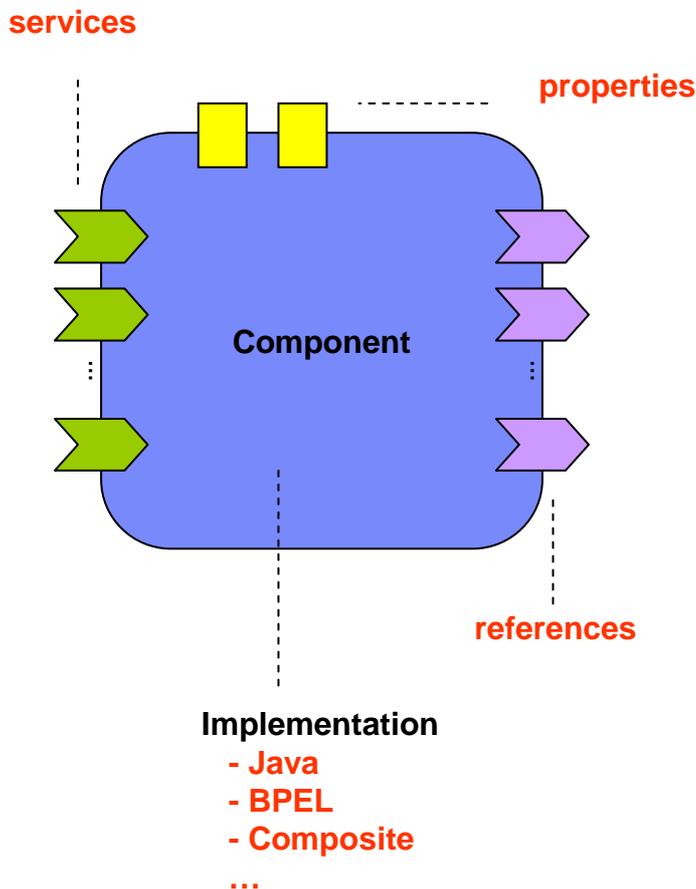
1106       The **component property** element has the following **child element**:

1107       **value :any (0..n)** - A property has **zero or more**, value elements that specify the value(s) of a  
1108       property that is defined using a XML Schema type. If a property is single-valued, the <value/>  
1109       subelement **MUST NOT** occur more than once. [\[ASM50032\]](#) A property <value/> subelement **MUST**  
1110       NOT be used when the @value attribute is used to specify the value for that property. [\[ASM50033\]](#)

## 1111 5.5 Example Component

1112

1113       The following figure shows the **component symbol** that is used to represent a component in an  
1114       assembly diagram.

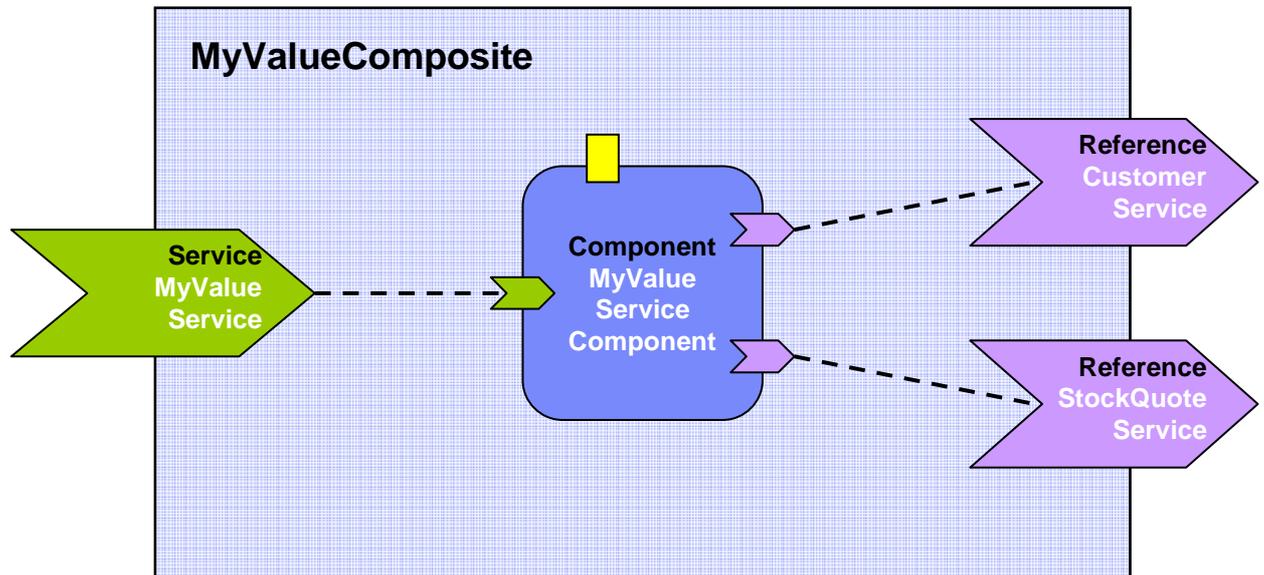


1115

1116       Figure 5: Component symbol

1117       The following figure shows the assembly diagram for the MyValueComposite containing the  
1118       MyValueServiceComponent.

1119



1120

1121

1122 *Figure 6: Assembly diagram for MyValueComposite*

1123

1124 The following snippet shows the MyValueComposite.composite file for the MyValueComposite  
 1125 containing the component element for the MyValueServiceComponent. A value is set for the  
 1126 property named currency, and the customerService and stockQuoteService references are  
 1127 promoted:

1128

```

1129 <?xml version="1.0" encoding="ASCII"?>
1130 <!-- MyValueComposite_1 example -->
1131 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1132 targetNamespace="http://foo.com"
1133 name="MyValueComposite" >
1134
1135 <service name="MyValueService" promote="MyValueServiceComponent" />
1136
1137 <component name="MyValueServiceComponent">
1138 <implementation.java
1139 class="services.myvalue.MyValueServiceImpl" />
1140 <property name="currency">EURO</property>
1141 <reference name="customerService" />
1142 <reference name="stockQuoteService" />
1143 </component>
1144
1145 <reference name="CustomerService"
1146 promote="MyValueServiceComponent/customerService" />
1147
1148 <reference name="StockQuoteService"
1149 promote="MyValueServiceComponent/stockQuoteService" />
1150
1151 </composite>

```

1152

1153 Note that the references of MyValueServiceComponent are explicitly declared only for purposes of  
 1154 clarity – the references are defined by the MyValueServiceImpl implementation and there is no  
 1155 need to redeclare them on the component unless the intention is to wire them or to override some  
 1156 aspect of them.

1157 The following snippet gives an example of the layout of a composite file if both the currency  
1158 property and the customerService reference of the MyValueServiceComponent are declared to be  
1159 multi-valued (many=true for the property and multiplicity=0..n or 1..n for the reference):

```
1160 <?xml version="1.0" encoding="ASCII"?>
1161 <!-- MyValueComposite_2 example -->
1162 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1163 targetNamespace="http://foo.com"
1164 name="MyValueComposite" >
1165
1166 <service name="MyValueService" promote="MyValueServiceComponent" />
1167
1168 <component name="MyValueServiceComponent">
1169 <implementation.java
1170 class="services.myvalue.MyValueServiceImpl" />
1171 <property name="currency">EURO</property>
1172 <property name="currency">Yen</property>
1173 <property name="currency">USDollar</property>
1174 <reference name="customerService"
1175 target="InternalCustomer/customerService" />
1176 <reference name="StockQuoteService" />
1177 </component>
1178
1179 ...
1180
1181 <reference name="CustomerService"
1182 promote="MyValueServiceComponent/customerService" />
1183
1184 <reference name="StockQuoteService"
1185 promote="MyValueServiceComponent/StockQuoteService" />
1186
1187 </composite>
```

1188  
1189 ....this assumes that the composite has another component called InternalCustomer (not shown)  
1190 which has a service to which the customerService reference of the MyValueServiceComponent is  
1191 wired as well as being promoted externally through the composite reference CustomerService.

## 6 Composite

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

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

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

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

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

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

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

- **name : NCName (1..1)** – the name of the composite. The form of a composite name is an XML QName, in the namespace identified by the targetNamespace attribute. A composite name must be unique within the namespace of the composite. [\[ASM60001\]](#)
- **targetNamespace : anyURI (0..1)** – an identifier for a target namespace into which the composite is declared
- **local : boolean (0..1)** – whether all the components within the composite all run in the same operating system process. @local="true" for a composite means that all the components within the composite **MUST** run in the same operating system process. [\[ASM60002\]](#) local="false", which is the default, means that different components within the composite can run in different operating system processes and they can even run on different nodes on a network.

- 1242 • **autowire : boolean (0..1)** – whether contained component references should be  
1243 autowired, as described in [the Autowire section](#). Default is false.
- 1244 • **constrainingType : QName (0..1)** – the name of a constrainingType. When specified,  
1245 the set of services, references and properties of the composite, plus related intents, is  
1246 constrained to the set defined by the constrainingType. See [the ConstrainingType Section](#)  
1247 for more details.
- 1248 • **requires : QName (0..n)** – a list of policy intents. See the [Policy Framework](#)  
1249 [specification \[10\]](#) for a description of this attribute.
- 1250 • **policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification](#)  
1251 [\[10\]](#) for a description of this attribute.

1252

1253 The **composite** element has the following **child elements**:

- 1254 • **service : CompositeService (0..n)** – see composite service section.
- 1255 • **reference : CompositeReference (0..n)** – see composite reference section.
- 1256 • **property : CompositeProperty (0..n)** – see composite property section.
- 1257 • **component : Component (0..n)** – see component section.
- 1258 • **wire : Wire (0..n)** – see composite wire section.
- 1259 • **include : Include (0..n)** – see composite include section

1260

1261 Components contain configured implementations which hold the business logic of the composite.  
1262 The components offer services and require references to other services. **Composite services**  
1263 define the public services provided by the composite, which can be accessed from outside the  
1264 composite. **Composite references** represent dependencies which the composite has on services  
1265 provided elsewhere, outside the composite. Wires describe the connections between component  
1266 services and component references within the composite. Included composites contribute the  
1267 elements they contain to the using composite.

1268 Composite services involve the **promotion** of one service of one of the components within the  
1269 composite, which means that the composite service is actually provided by one of the components  
1270 within the composite. Composite references involve the **promotion** of one or more references of  
1271 one or more components. Multiple component references can be promoted to the same composite  
1272 reference, as long as all the component references are compatible with one another. Where  
1273 multiple component references are promoted to the same composite reference, then they all share  
1274 the same configuration, including the same target service(s).

1275 Composite services and composite references can use the configuration of their promoted services  
1276 and references respectively (such as Bindings and Policy Sets). Alternatively composite services  
1277 and composite references can override some or all of the configuration of the promoted services  
1278 and references, through the configuration of bindings and other aspects of the composite service  
1279 or reference.

1280 Component services and component references can be promoted to composite services and  
1281 references and also be wired internally within the composite at the same time. For a reference,  
1282 this only makes sense if the reference supports a multiplicity greater than 1.

1283

## 1284 6.1 Service

1285 The **services of a composite** are defined by promoting services defined by components  
1286 contained in the composite. A component service is promoted by means of a composite **service**  
1287 **element**.

1288 A composite service is represented by a **service element** which is a child of the composite  
1289 element. There can be **zero or more** service elements in a composite. The following snippet  
1290 shows the pseudo-schema for a service child element:

```
1291
1292 <?xml version="1.0" encoding="ASCII"?>
1293 <!-- Composite Service schema snippet -->
1294 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
1295 ...
1296 <service name="xs:NCName" promote="xs:anyURI"
1297 requires="list of xs:QName"? policySets="list of xs:QName"?>*
1298 <interface ... />?
1299 <operation name="xs:NCName" requires="list of xs:QName"?
1300 policySets="list of xs:QName"?/>*
1301 <binding ... />*
1302 <callback?>
1303 <binding ... />+
1304 </callback>
1305 </service>
1306 ...
1307 </composite>
1308
```

1309 The **composite service** element has the following **attributes**:

- 1310 • **name : NCName (1..1)** – the name of the service. The name of a composite <service/>  
1311 element MUST be unique across all the composite services in the composite. [ASM60003]  
1312 The name of the composite service can be different from the name of the promoted  
1313 component service.
- 1314 • **promote : anyURI (1..1)** – identifies the promoted service, the value is of the form  
1315 <component-name>/<service-name>. The service name is optional if the target  
1316 component only has one service. The same component service can be promoted by more  
1317 than one composite service. A composite <service/> element's promote attribute MUST  
1318 identify one of the component services within that composite. [ASM60004]
- 1319 • **requires : QName (0..n)** – a list of required policy intents. See the [Policy Framework](#)  
1320 [specification \[10\]](#) for a description of this attribute. Specified **required intents** add to or  
1321 further qualify the required intents defined by the promoted component service.
- 1322 • **policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification](#)  
1323 [\[10\]](#) for a description of this attribute.

1324

1325 The **composite service** element has the following **child elements**, whatever is not specified is  
1326 defaulted from the promoted component service.

- 1327 • **interface : Interface (0..1)** - If a composite service **interface** is specified it must be the  
1328 same or a compatible subset of the interface provided by the promoted component  
1329 service, i.e. provide a subset of the operations defined by the component service.  
1330 [ASM60005] The interface is described by **zero or one interface element** which is a  
1331 child element of the service element. For details on the interface element see [the Interface](#)  
1332 [section](#).
- 1333 • **operation: Operation (0..n)** - Zero or more operation elements. These elements are  
1334 used to describe characteristics of individual operations within the interface. For a detailed  
1335 description of the operation element, see [the Policy Framework specification \[SCA Policy\]](#).
- 1336 • **binding : Binding (0..n)** - If bindings are specified they **override** the bindings defined  
1337 for the promoted component service from the composite service perspective. The bindings  
1338 defined on the component service are still in effect for local wires within the composite  
1339 that target the component service. A service element has zero or more **binding elements**

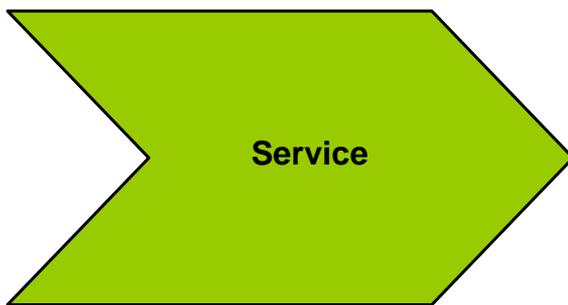
1340 as children. Details of the binding element are described in the [Bindings section](#). For more  
1341 details on wiring see [the Wiring section](#).

- 1342 • **callback (0..1) / binding : Binding (1..n)** - A service element has an optional **callback**  
1343 element used if the interface has a callback defined, which has one or more **binding**  
1344 elements as children. The **callback** and its binding child elements are specified if there is  
1345 a need to have binding details used to handle callbacks. If the callback element is not  
1346 present, the behaviour is runtime implementation dependent.

1347

## 1348 6.1.1 Service Examples

1349 The following figure shows the service symbol that used to represent a service in an assembly  
1350 diagram:

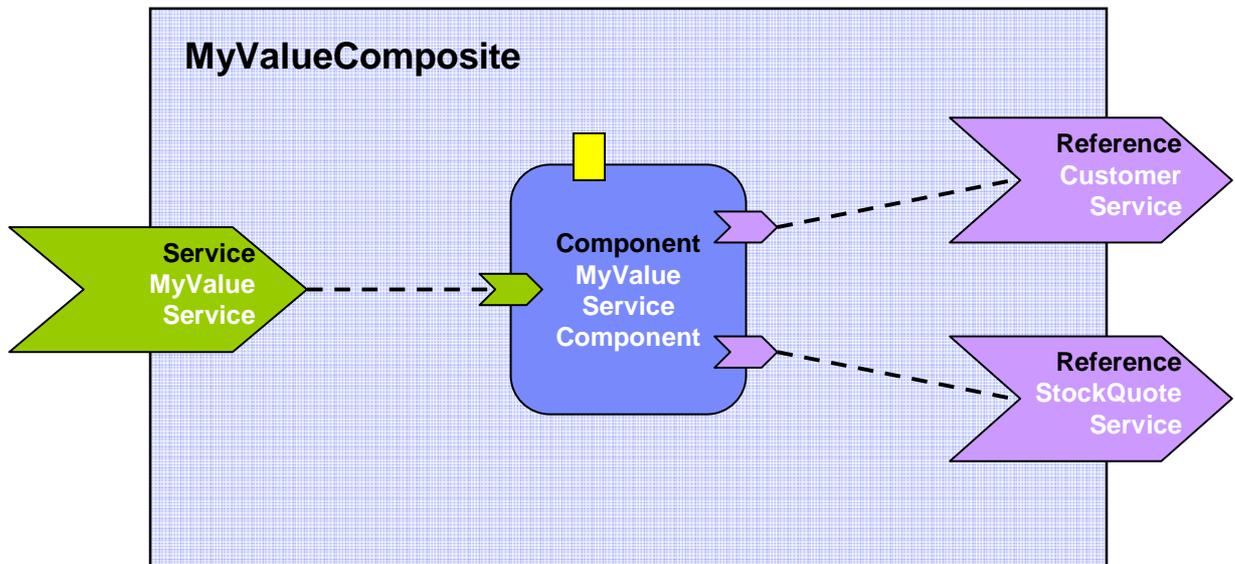


1351

1352 *Figure 7: Service symbol*

1353

1354 The following figure shows the assembly diagram for the MyValueComposite containing the service  
1355 MyValueService.



1356

1357 *Figure 8: MyValueComposite showing Service*

1358

1359 The following snippet shows the MyValueComposite.composite file for the MyValueComposite  
1360 containing the service element for the MyValueService, which is a promote of the service offered  
1361 by the MyValueServiceComponent. The name of the promoted service is omitted since  
1362 MyValueServiceComponent offers only one service. The composite service MyValueService is  
1363 bound using a Web service binding.

```

1364
1365 <?xml version="1.0" encoding="ASCII"?>
1366 <!-- MyValueComposite_4 example -->
1367 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1368 targetNamespace="http://foo.com"
1369 name="MyValueComposite" >
1370
1371 ...
1372
1373 <service name="MyValueService" promote="MyValueServiceComponent">
1374 <interface.java interface="services.myvalue.MyValueService"/>
1375 <binding.ws port="http://www.myvalue.org/MyValueService#
1376 wsdl.endpoint(MyValueService/MyValueServiceSOAP)"/>
1377 </service>
1378
1379 <component name="MyValueServiceComponent">
1380 <implementation.java
1381 class="services.myvalue.MyValueServiceImpl"/>
1382 <property name="currency">EURO</property>
1383 <service name="MyValueService"/>
1384 <reference name="customerService"/>
1385 <reference name="StockQuoteService"/>
1386 </component>
1387
1388 ...
1389
1390 </composite>
1391

```

## 1392 6.2 Reference

1393 The **references of a composite** are defined by **promoting** references defined by components  
1394 contained in the composite. Each promoted reference indicates that the component reference  
1395 needs to be resolved by services outside the composite. A component reference is promoted using  
1396 a composite **reference element**.

1397 A composite reference is represented by a **reference element** which is a child of a composite  
1398 element. There can be **zero or more** reference elements in a composite. The following snippet  
1399 shows the composite schema with the schema for a **reference** element.

```

1400
1401 <?xml version="1.0" encoding="ASCII"?>
1402 <!-- Composite Reference schema snippet -->
1403 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
1404 ...
1405 <reference name="xs:NCName" target="list of xs:anyURI"?
1406 promote="list of xs:anyURI" wiredByImpl="xs:boolean"?
1407 multiplicity="0..1 or 1..1 or 0..n or 1..n"?
1408 requires="list of xs:QName"? policySets="list of xs:QName"?>*
1409 <interface ... />?
1410 <operation name="xs:NCName" requires="list of xs:QName"?
1411 policySets="list of xs:QName"?/>*
1412 <binding ... />*
1413 <callback?
1414 <binding ... />+
1415 </callback>
1416 </reference>
1417 ...

```

1418 </composite>

1419  
1420

1421 The **composite reference** element has the following **attributes**:

- 1422 • **name : NCName (1..1)** – the name of the reference. The name of a composite  
1423 <reference/> element MUST be unique across all the composite references in the  
1424 composite. [ASM60006] The name of the composite reference can be different then the  
1425 name of the promoted component reference.
- 1426 • **promote : anyURI (1..n)** – identifies one or more promoted component references. The  
1427 value is a list of values of the form <component-name>/<reference-name> separated by  
1428 spaces. The specification of the reference name is optional if the component has only one  
1429 reference. Each of the URIs declared by a composite reference's @promote attribute MUST  
1430 identify a component reference within the composite. [ASM60007]

1431 The same component reference can be promoted more than once, using different  
1432 composite references, but only if the multiplicity defined on the component reference is  
1433 0..n or 1..n. The multiplicity on the composite reference can restrict accordingly.

1434 Where a composite reference promotes two or more component references:

- 1435 • the interfaces of the component references promoted by a composite reference  
1436 MUST be the same, or if the composite reference itself declares an interface then  
1437 all the component reference interfaces must be compatible with the composite  
1438 reference interface. Compatible means that the component reference interface is  
1439 the same or is a strict subset of the composite reference interface. [ASM60008]
- 1440 • the intents declared on a composite reference and on the component references  
1441 which it promotes MUST NOT be mutually exclusive. [ASM60009] The intents  
1442 which apply to the composite reference in this case are the union of the required  
1443 intents specified for each of the promoted component references plus any intents  
1444 declared on the composite reference itself. If any intents in the set which apply to  
1445 a composite reference are mutually exclusive then the SCA runtime MUST raise an  
1446 error. [ASM60010]

- 1447 • **requires : QName (0..n)** – a list of required policy intents. See the [Policy Framework](#)  
1448 [specification \[10\]](#) for a description of this attribute. Specified **required intents** add to or  
1449 further qualify the required intents defined for the promoted component reference.

- 1450 • **policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification](#)  
1451 [\[10\]](#) for a description of this attribute.

- 1452 • **multiplicity : 0..1|1..1|0..n|1..n (1..1)** - Defines the number of wires that can  
1453 connect the reference to target services. The multiplicity can have the following values

- 1454 ○ 0..1 – zero or one wire can have the reference as a source
- 1455 ○ 1..1 – one wire can have the reference as a source
- 1456 ○ 0..n - zero or more wires can have the reference as a source
- 1457 ○ 1..n – one or more wires can have the reference as a source

1458 The value specified for the **multiplicity** attribute of a composite reference MUST be  
1459 compatible with the multiplicity specified on each of the promoted component references,  
1460 i.e. the multiplicity has to be equal or further restrict. So multiplicity 0..1 can be used  
1461 where the promoted component reference has multiplicity 0..n, multiplicity 1..1 can be  
1462 used where the promoted component reference has multiplicity 0..n or 1..n and  
1463 multiplicity 1..n can be used where the promoted component reference has multiplicity  
1464 0..n., However, a composite reference of multiplicity 0..n or 1..n cannot be used to  
1465 promote a component reference of multiplicity 0..1 or 1..1 respectively. [ASM60011]

- 1466 • **target : anyURI (0..n)** – a list of one or more of target service URI's, depending on  
1467 multiplicity setting. Each value wires the reference to a service in a composite that uses

1468 the composite containing the reference as an implementation for one of its components. For  
1469 more details on wiring see [the section on Wires](#).

1470 • **wiredByImpl : boolean (0..1)** – a boolean value, "false" by default, which indicates that  
1471 the implementation wires this reference dynamically. If set to "true" it indicates that the  
1472 target of the reference is set at runtime by the implementation code (eg by the code  
1473 obtaining an endpoint reference by some means and setting this as the target of the  
1474 reference through the use of programming interfaces defined by the relevant Client and  
1475 Implementation specification). If "true" is set, then the reference should not be wired  
1476 statically within a using composite, but left unwired.

1477  
1478 The **composite reference** element has the following **child elements**, whatever is not specified is  
1479 defaulted from the promoted component reference(s).

1480 • **interface : Interface (0..1) - zero or one interface element** which declares an  
1481 interface for the composite reference. If a composite reference has an **interface** specified,  
1482 it **MUST** provide an interface which is the same or which is a compatible superset of the  
1483 interface(s) declared by the promoted component reference(s), i.e. provide a superset of  
1484 the operations in the interface defined by the component for the reference. [ASM60012] If  
1485 no interface is declared on a composite reference, the interface from one of its promoted  
1486 component references is used, which **MUST** be the same as or a compatible superset of  
1487 the interface(s) declared by the promoted component reference(s).  
1488 [ASM60013] For details on the interface element see [the Interface section](#).

1489 • **operation: Operation (0..n)** - Zero or more operation elements. These elements are  
1490 used to describe characteristics of individual operations within the interface. For a detailed  
1491 description of the operation element, see [the Policy Framework specification](#) [SCA Policy].

1492 • **binding : Binding (0..n)** - A reference element has zero or more **binding elements** as  
1493 children. If one or more **bindings** are specified they **override** any and all of the bindings  
1494 defined for the promoted component reference from the composite reference perspective.  
1495 The bindings defined on the component reference are still in effect for local wires within  
1496 the composite that have the component reference as their source. Details of the binding  
1497 element are described in the [Bindings section](#). For more details on wiring see [the section](#)  
1498 [on Wires](#).

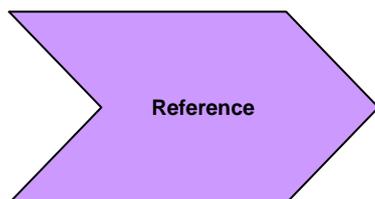
1499 A reference identifies zero or more target services which satisfy the reference. This can be  
1500 done in a number of ways, which are fully described in section "5.3.1 Specifying the  
1501 [Target Service\(s\) for a Reference](#)".

1502 • **callback (0..1) / binding : Binding (1..n)** - A **reference** element has an optional  
1503 **callback** element used if the interface has a callback defined, which has one or more  
1504 **binding** elements as children. The **callback** and its binding child elements are specified if  
1505 there is a need to have binding details used to handle callbacks. If the callback element is  
1506 not present, the behaviour is runtime implementation dependent.

1507

## 1508 6.2.1 Example Reference

1509 The following figure shows the reference symbol that is used to represent a reference in an  
1510 assembly diagram.

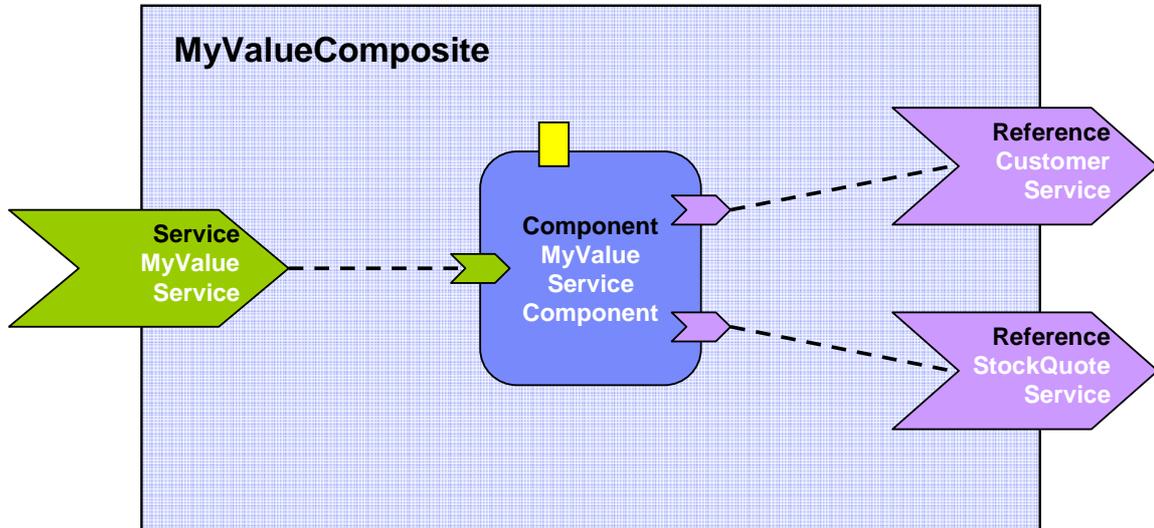


1511

1512 *Figure 9: Reference symbol*

1513

1514 The following figure shows the assembly diagram for the MyValueComposite containing the  
 1515 reference CustomerService and the reference StockQuoteService.  
 1516



1517  
 1518 *Figure 10: MyValueComposite showing References*

1519  
 1520 The following snippet shows the MyValueComposite.composite file for the MyValueComposite  
 1521 containing the reference elements for the CustomerService and the StockQuoteService. The  
 1522 reference CustomerService is bound using the SCA binding. The reference StockQuoteService is  
 1523 bound using the Web service binding. The endpoint addresses of the bindings can be specified, for  
 1524 example using the binding *uri* attribute (for details see the [Bindings](#) section), or overridden in an  
 1525 enclosing composite. Although in this case the reference StockQuoteService is bound to a Web  
 1526 service, its interface is defined by a Java interface, which was created from the WSDL portType of  
 1527 the target web service.

```

1528
1529 <?xml version="1.0" encoding="ASCII"?>
1530 <!-- MyValueComposite_3 example -->
1531 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1532 targetNamespace="http://foo.com"
1533 name="MyValueComposite" >
1534
1535 ...
1536
1537 <component name="MyValueServiceComponent">
1538 <implementation.java
1539 class="services.myvalue.MyValueServiceImpl"/>
1540 <property name="currency">EURO</property>
1541 <reference name="customerService"/>
1542 <reference name="StockQuoteService"/>
1543 </component>
1544
1545 <reference name="CustomerService"
1546 promote="MyValueServiceComponent/customerService">
1547 <interface.java interface="services.customer.CustomerService"/>
1548 <!-- The following forces the binding to be binding.sca -->
1549 <!-- whatever is specified by the component reference or by -->
1550 <!-- the underlying implementation -->
1551 <binding.sca/>

```

```

1552 </reference>
1553
1554 <reference name="StockQuoteService"
1555 promote="MyValueServiceComponent/StockQuoteService">
1556 <interface.java
1557 interface="services.stockquote.StockQuoteService"/>
1558 <binding.ws port="http://www.stockquote.org/StockQuoteService#
1559 wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
1560 </reference>
1561
1562 ...
1563
1564 </composite>
1565

```

## 1566 6.3 Property

1567 **Properties** allow for the configuration of an implementation with externally set data values. A  
1568 composite can declare zero or more properties. Each property has a type, which may be either  
1569 simple or complex. An implementation can also define a default value for a property. Properties  
1570 can be configured with values in the components that use the implementation.

1571 The declaration of a property in a composite follows the form described in the following schema  
1572 snippet:

```

1573
1574 <?xml version="1.0" encoding="ASCII"?>
1575 <!-- Composite Property schema snippet -->
1576 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
1577 ...
1578 <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
1579 requires="list of xs:QName"?
1580 policySets="list of xs:QName"?
1581 many="xs:boolean"? mustSupply="xs:boolean"?>*
1582 default-property-value?
1583 </property>
1584 ...
1585 </composite>
1586

```

1587 The **composite property** element has the following **attributes**:

- 1588 ▪ **name : NCName (1..1)** - the name of the property. The name attribute of a composite  
1589 property MUST be unique amongst the properties of the same composite. [ASM60014]
- 1590 ▪ one of **(1..1)**:
  - 1591 ○ **type : QName** - the type of the property - the qualified name of an XML schema  
1592 type
  - 1593 ○ **element : QName** - the type of the property defined as the qualified name of an  
1594 XML schema global element - the type is the type of the global element
- 1595 ▪ **many : boolean (0..1)** - whether the property is single-valued (false) or multi-valued  
1596 (true). The default is **false**. In the case of a multi-valued property, it is presented to the  
1597 implementation as a collection of property values.
- 1598 ▪ **mustSupply : boolean (0..1)** - whether the property value has to be supplied by the  
1599 component that uses the composite - when mustSupply="true" the component has to  
1600 supply a value since the composite has no default value for the property. A default-  
1601 property-value is only worth declaring when mustSupply="false" (the default setting for  
1602 the mustSupply attribute), since the implication of a default value is that it is used only  
1603 when a value is not supplied by the using component.

- 1604       ▪ **requires : QName (0..n)** - a list of policy intents. See the [Policy Framework specification](#)  
1605       [\[10\]](#) for a description of this attribute.
- 1606       ▪ **policySets : QName (0..n)** - a list of policy sets. See the [Policy Framework specification](#)  
1607       [\[10\]](#) for a description of this attribute.

1608

1609       The property element may contain an optional **default-property-value**, which provides default  
1610       value for the property. The form of the default property value is as described [in the section on](#)  
1611       [Component Property](#).

1612

1613       Implementation types other than **composite** can declare properties in an implementation-  
1614       dependent form (eg annotations within a Java class), or through a property declaration of exactly  
1615       the form described above in a componentType file.

1616       Property values can be configured when an implementation is used by a component. The form of  
1617       the property configuration is shown in [the section on Components](#).

### 1618   6.3.1 Property Examples

1619

1620       For the following example of Property declaration and value setting, the following complex type is  
1621       used as an example:

```
1622 <xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
1623 targetNamespace="http://foo.com/"
1624 xmlns:tns="http://foo.com/">
1625 <!-- ComplexProperty schema -->
1626 <xsd:element name="fooElement" type="MyComplexType"/>
1627 <xsd:complexType name="MyComplexType">
1628 <xsd:sequence>
1629 <xsd:element name="a" type="xsd:string"/>
1630 <xsd:element name="b" type="anyURI"/>
1631 </xsd:sequence>
1632 <attribute name="attr" type="xsd:string" use="optional"/>
1633 </xsd:complexType>
1634 </xsd:schema>
```

1635

1636       The following composite demonstrates the declaration of a property of a complex type, with a  
1637       default value, plus it demonstrates the setting of a property value of a complex type within a  
1638       component:

```
1639 <?xml version="1.0" encoding="ASCII"?>
1640 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1641 xmlns:foo="http://foo.com"
1642 targetNamespace="http://foo.com"
1643 name="AccountServices">
1644 <!-- AccountServices Example1 -->
1645 ...
1646 ...
1647 ...
1648 <property name="complexFoo" type="foo:MyComplexType">
1649 <value>
1650 <foo:a>AValue</foo:a>
1651 <foo:b>InterestingURI</foo:b>
1652 </value>
1653 </property>
1654 </composite>
```

```

1655 <component name="AccountServiceComponent">
1656 <implementation.java class="foo.AccountServiceImpl"/>
1657 <property name="complexBar" source="$complexFoo"/>
1658 <reference name="accountDataService"
1659 target="AccountDataServiceComponent"/>
1660 <reference name="stockQuoteService" target="StockQuoteService"/>
1661 </component>
1662
1663 ...
1664
1665 </composite>
1666

```

1667 In the declaration of the property named **complexFoo** in the composite **AccountServices**, the
1668 property is defined to be of type **foo:MyComplexType**. The namespace **foo** is declared in the
1669 composite and it references the example XSD, where MyComplexType is defined. The declaration
1670 of complexFoo contains a default value. This is declared as the content of the property element.
1671 In this example, the default value consists of the element **value** which is required to be of type
1672 foo:MyComplexType and its two child elements <foo:a> and <foo:b>, following the definition of
1673 MyComplexType.

1674 In the component **AccountServiceComponent**, the component sets the value of the property
1675 **complexBar**, declared by the implementation configured by the component. In this case, the
1676 type of complexBar is foo:MyComplexType. The example shows that the value of the complexBar
1677 property is set from the value of the complexFoo property – the **source** attribute of the property
1678 element for complexBar declares that the value of the property is set from the value of a property
1679 of the containing composite. The value of the source attribute is **\$complexFoo**, where
1680 complexFoo is the name of a property of the composite. This value implies that the whole of the
1681 value of the source property is used to set the value of the component property.

1682 The following example illustrates the setting of the value of a property of a simple type (a string)
1683 from **part** of the value of a property of the containing composite which has a complex type:

```

1684 <?xml version="1.0" encoding="ASCII"?>
1685 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1686 xmlns:foo="http://foo.com"
1687 targetNamespace="http://foo.com"
1688 name="AccountServices">
1689 <!-- AccountServices Example2 -->
1690
1691 ...
1692
1693 <property name="complexFoo" type="foo:MyComplexType">
1694 <value>
1695 <foo:a>AValue</foo:a>
1696 <foo:b>InterestingURI</foo:b>
1697 </value>
1698 </property>
1699
1700 <component name="AccountServiceComponent">
1701 <implementation.java class="foo.AccountServiceImpl"/>
1702 <property name="currency" source="$complexFoo/a"/>
1703 <reference name="accountDataService"
1704 target="AccountDataServiceComponent"/>
1705 <reference name="stockQuoteService" target="StockQuoteService"/>
1706 </component>
1707
1708 ...
1709
1710 </composite>
1711

```

1712 In this example, the component **AccountServiceComponent** sets the value of a property called  
1713 **currency**, which is of type string. The value is set from a property of the composite  
1714 **AccountServices** using the source attribute set to **\$complexFoo/a**. This is an XPath expression  
1715 that selects the property name **complexFoo** and then selects the value of the **a** subelement of  
1716 the value of complexFoo. The "a" subelement is a string, matching the type of the currency  
1717 property.

1718 Further examples of declaring properties and setting property values in a component follow:

1719 Declaration of a property with a simple type and a default value:

```
1720 <property name="SimpleTypeProperty" type="xsd:string">
1721 MyValue
1722 </property>
```

1723

1724 Declaration of a property with a complex type and a default value:

```
1725 <property name="complexFoo" type="foo:MyComplexType">
1726 <value>
1727 <foo:a>AValue</foo:a>
1728 <foo:b>InterestingURI</foo:b>
1729 </value>
1730 </property>
```

1731

1732 Declaration of a property with a global element type:

```
1733 <property name="elementFoo" element="foo:fooElement">
1734 <foo:fooElement>
1735 <foo:a>AValue</foo:a>
1736 <foo:b>InterestingURI</foo:b>
1737 </foo:fooElement>
1738 </property>
```

1739

## 1740 6.4 Wire

1741 **SCA wires** within a composite connect **source component references** to **target component**  
1742 **services**.

1743 One way of defining a wire is by **configuring a reference of a component using its target**  
1744 **attribute**. The reference element is configured with the wire-target-URI of the service(s) that  
1745 resolve the reference. Multiple target services are valid when the reference has a multiplicity of  
1746 0..n or 1..n.

1747 An alternative way of defining a Wire is by means of a **wire element** which is a child of the  
1748 composite element. There can be **zero or more** wire elements in a composite. This alternative  
1749 method for defining wires is useful in circumstances where separation of the wiring from the  
1750 elements the wires connect helps simplify development or operational activities. An example is  
1751 where the components used to build a domain are relatively static but where new or changed  
1752 applications are created regularly from those components, through the creation of new assemblies  
1753 with different wiring. Deploying the wiring separately from the components allows the wiring to  
1754 be created or modified with minimum effort.

1755 Note that a Wire specified via a wire element is equivalent to a wire specified via the target  
1756 attribute of a reference. The rule which forbids mixing of wires specified with the target attribute  
1757 with the specification of endpoints in binding subelements of the reference also applies to wires  
1758 specified via separate wire elements.

1759 The following snippet shows the composite schema with the schema for the reference elements of  
1760 components and composite services and the wire child element:

1761

```

1762 <?xml version="1.0" encoding="ASCII"?>
1763 <!-- Wires schema snippet -->
1764 <composite ...>
1765 ...
1766 <wire source="xs:anyURI" target="xs:anyURI" replace="xs:boolean"?/>*
1767 ...
1768 </composite>
1769

```

1770 The **reference element of a component** and the **reference element of a service** has a list of  
1771 one or more of the following **wire-target-URI** values for the target, with multiple values  
1772 separated by a space:

- 1773
- `<component-name>/<service-name>`
    - where the target is a service of a component. The specification of the service name is optional if the target component only has one service with a compatible interface

1777

1778 The **wire element** has the following attributes:

- 1779
- **source (1..1)** – names the source component reference. Valid URI schemes are:
    - `<component-name>/<reference-name>`
      - where the source is a component reference. The specification of the reference name is optional if the source component only has one reference
  - **target (1..1)** – names the target component service. Valid URI schemes are
    - `<component-name>/<service-name>`
      - where the target is a service of a component. The specification of the service name is optional if the target component only has one service with a compatible interface
  - **replace (0..1)** – a boolean value, with the default of "false". When a wire element has @replace="false", the wire is added to the set of wires which apply to the reference identified by the @source attribute. When a wire element has @replace="true", the wire is added to the set of wires which apply to the reference identified by the @source attribute - but any wires for that reference specified by means of the @target attribute of the reference are removed from the set of wires which apply to the reference.

1788  
1789  
1790  
1791  
1792  
1793  
1794  
1795  
1796  
1797  
1798  
 In other words, if any `<wire/>` element with @replace="true" is used for a particular reference, the value of the @target attribute on the reference is ignored - and this permits existing wires on the reference to be overridden by separate configuration, if required, where the reference is on a component at the Domain level.

1799 For a composite used as a component implementation, wires can only link sources and targets  
1800 that are contained in the same composite (irrespective of which file or files are used to describe  
1801 the composite). Wiring to entities outside the composite is done through services and references  
1802 of the composite with wiring defined by the next higher composite.

1803 A wire may only connect a source to a target if the target implements an interface that is  
1804 compatible with the interface required by the source. The source and the target are compatible if:

- 1805
1. the source interface and the target interface of a wire MUST either both be remotable or else both be local [ASM60015]
  - 1807 2. the operations on the target interface of a wire MUST be the same as or be a superset of  
 1808 the operations in the interface specified on the source [ASM60016]
  - 1809 3. compatibility between the source interface and the target interface for a wire for the  
 1810 individual operations is defined as compatibility of the signature, that is operation name,  
 1811 input types, and output types MUST be the same. [ASM60017]

- 1812 4. the order of the input and output types for operations in the source interface and the  
 1813 target interface of a wire also MUST be the same. [ASM60018]
- 1814 5. the set of Faults and Exceptions expected by each operation in the source interface MUST  
 1815 be the same or be a superset of those specified by the target interface. [ASM60019]
- 1816 6. other specified attributes of the source interface and the target interface of a wire MUST  
 1817 match, including Scope and Callback interface [ASM60020]

1818 A Wire can connect between different interface languages (eg. Java interfaces and WSDL  
 1819 portTypes) in either direction, as long as the operations defined by the two interface types are  
 1820 equivalent. They are equivalent if the operation(s), parameter(s), return value(s) and  
 1821 faults/exceptions map to each other.

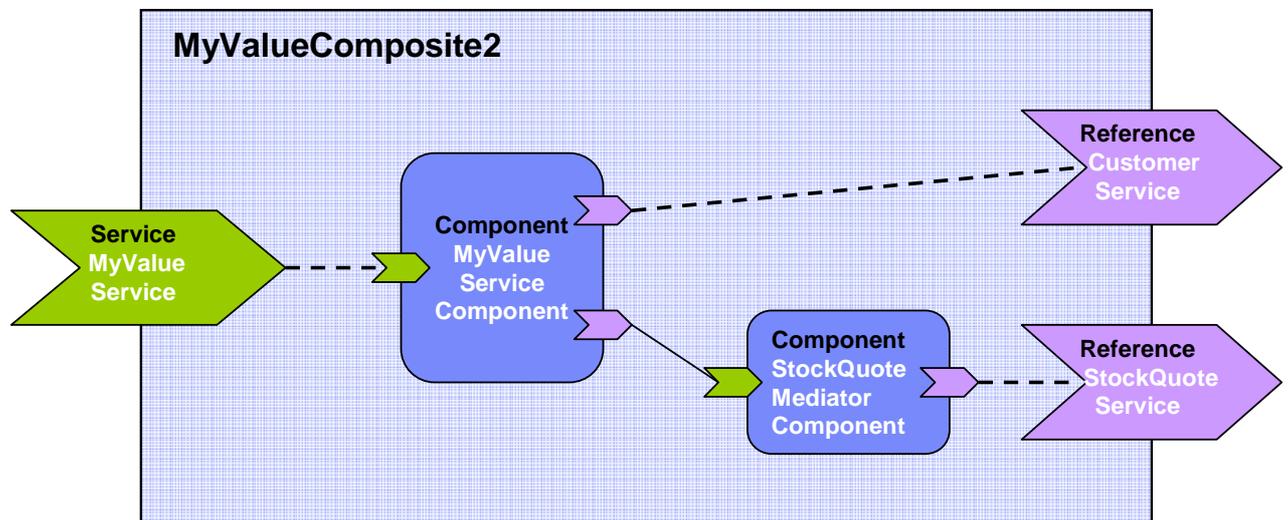
1822 Service clients cannot (portably) ask questions at runtime about additional interfaces that are  
 1823 provided by the implementation of the service (e.g. the result of "instance of" in Java is non  
 1824 portable). It is valid for an SCA implementation to have proxies for all wires, so that, for example,  
 1825 a reference object passed to an implementation may only have the business interface of the  
 1826 reference and may not be an instance of the (Java) class which is used to implement the target  
 1827 service, even where the interface is local and the target service is running in the same process.

1828 **Note:** It is permitted to deploy a composite that has references that are not wired. For the case of  
 1829 an un-wired reference with multiplicity 1..1 or 1..n the deployment process provided by an SCA  
 1830 runtime SHOULD issue a warning. [ASM60021]

1831

## 1832 6.4.1 Wire Examples

1833 The following figure shows the assembly diagram for the MyValueComposite2 containing wires  
 1834 between service, components and references.



1835

1836 *Figure 11: MyValueComposite2 showing Wires*

1837

1838 The following snippet shows the MyValueComposite2.composite file for the MyValueComposite2  
 1839 containing the configured component and service references. The service MyValueService is wired  
 1840 to the MyValueServiceComponent, using an explicit <wire/> element. The  
 1841 MyValueServiceComponent's customerService reference is wired to the composite's  
 1842 CustomerService reference. The MyValueServiceComponent's stockQuoteService reference is  
 1843 wired to the StockQuoteMediatorComponent, which in turn has its reference wired to the  
 1844 StockQuoteService reference of the composite.

1845

1846 `<?xml version="1.0" encoding="ASCII"?>`

```

1847 <!-- MyValueComposite Wires examples -->
1848 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1849 targetNamespace="http://foo.com"
1850 name="MyValueComposite2" >
1851
1852 <service name="MyValueService" promote="MyValueServiceComponent">
1853 <interface.java interface="services.myvalue.MyValueService"/>
1854 <binding.ws port="http://www.myvalue.org/MyValueService#
1855 wsdl.endpoint(MyValueService/MyValueServiceSOAP)"/>
1856 </service>
1857
1858 <component name="MyValueServiceComponent">
1859 <implementation.java
1860 class="services.myvalue.MyValueServiceImpl"/>
1861 <property name="currency">EURO</property>
1862 <service name="MyValueService"/>
1863 <reference name="customerService"/>
1864 <reference name="stockQuoteService"/>
1865 </component>
1866
1867 <wire source="MyValueServiceComponent/stockQuoteService"
1868 target="StockQuoteMediatorComponent"/>
1869
1870 <component name="StockQuoteMediatorComponent">
1871 <implementation.java class="services.myvalue.SQMediatorImpl"/>
1872 <property name="currency">EURO</property>
1873 <reference name="stockQuoteService"/>
1874 </component>
1875
1876 <reference name="CustomerService"
1877 promote="MyValueServiceComponent/customerService">
1878 <interface.java interface="services.customer.CustomerService"/>
1879 <binding.sca/>
1880 </reference>
1881
1882 <reference name="StockQuoteService"
1883 promote="StockQuoteMediatorComponent">
1884 <interface.java
1885 interface="services.stockquote.StockQuoteService"/>
1886 <binding.ws port="http://www.stockquote.org/StockQuoteService#
1887 wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
1888 </reference>
1889
1890 </composite>
1891

```

## 1892 6.4.2 Autowire

1893 SCA provides a feature named **Autowire**, which can help to simplify the assembly of composites.  
1894 Autowire enables component references to be automatically wired to component services which  
1895 will satisfy those references, without the need to create explicit wires between the references and  
1896 the services. When the autowire feature is used, a component reference which is not promoted  
1897 and which is not explicitly wired to a service within a composite is automatically wired to a target  
1898 service within the same composite. Autowire works by searching within the composite for a  
1899 service interface which matches the interface of the references.

1900 The autowire feature is not used by default. Autowire is enabled by the setting of an autowire  
1901 attribute to "true". Autowire is disabled by setting of the autowire attribute to "false" The autowire  
1902 attribute can be applied to any of the following elements within a composite:

- 1903           • reference
  - 1904           • component
  - 1905           • composite
- 1906           Where an element does not have an explicit setting for the autowire attribute, it inherits the  
1907           setting from its parent element. Thus a reference element inherits the setting from its containing  
1908           component. A component element inherits the setting from its containing composite. Where  
1909           there is no setting on any level, autowire="false" is the default.
- 1910           As an example, if a composite element has autowire="true" set, this means that autowiring is  
1911           enabled for all component references within that composite. In this example, autowiring can be  
1912           turned off for specific components and specific references through setting autowire="false" on the  
1913           components and references concerned.
- 1914           For each component reference for which autowire is enabled, the the SCA runtime **MUST** search  
1915           within the composite for target services which are compatible with the reference. [ASM60022]  
1916           "Compatible" here means:
- 1917           • the target service interface **MUST** be a compatible superset of the reference interface  
1918           when using autowire to wire a reference (as defined in the section on Wires) [ASM60023]
  - 1919           • the intents, and policies applied to the service **MUST** be compatible with those on the  
1920           reference when using autowire to wire a reference – so that wiring the reference to the  
1921           service will not cause an error due to policy mismatch [ASM60024] (see [the Policy](#)  
1922           [Framework specification \[10\]](#) for details)
- 1923           If the search finds **1 or more** valid target service for a particular reference, the action taken  
1924           depends on the multiplicity of the reference:
- 1925           • for an autowire reference with multiplicity 0..1 or 1..1, the SCA runtime **MUST** wire the  
1926           reference to one of the set of valid target services chosen from the set in a runtime-  
1927           dependent fashion [ASM60025]
  - 1928           • for an autowire reference with multiplicity 0..n or 1..n, the reference **MUST** be wired to all  
1929           of the set of valid target services [ASM60026]
- 1930           If the search finds **no** valid target services for a particular reference, the action taken depends on  
1931           the multiplicity of the reference:
- 1932           • for an autowire reference with multiplicity 0..1 or 0..n, if the SCA runtime finds no valid  
1933           target service, there is no problem – no services are wired and the SCA runtime **MUST**  
1934           NOT raise an error [ASM60027]
  - 1935           • for an autowire reference with multiplicity 1..1 or 1..n, if the SCA runtime finds no valid  
1936           target services an error **MUST** be raised by the SCA runtime since the reference is  
1937           intended to be wired [ASM60028]
- 1938
- 1939           **6.4.3 Autowire Examples**
- 1940           This example demonstrates two versions of the same composite – the first version is done using  
1941           explicit wires, with no autowiring used, the second version is done using autowire. In both cases  
1942           the end result is the same – the same wires connect the references to the services.
- 1943           First, here is a diagram for the composite:

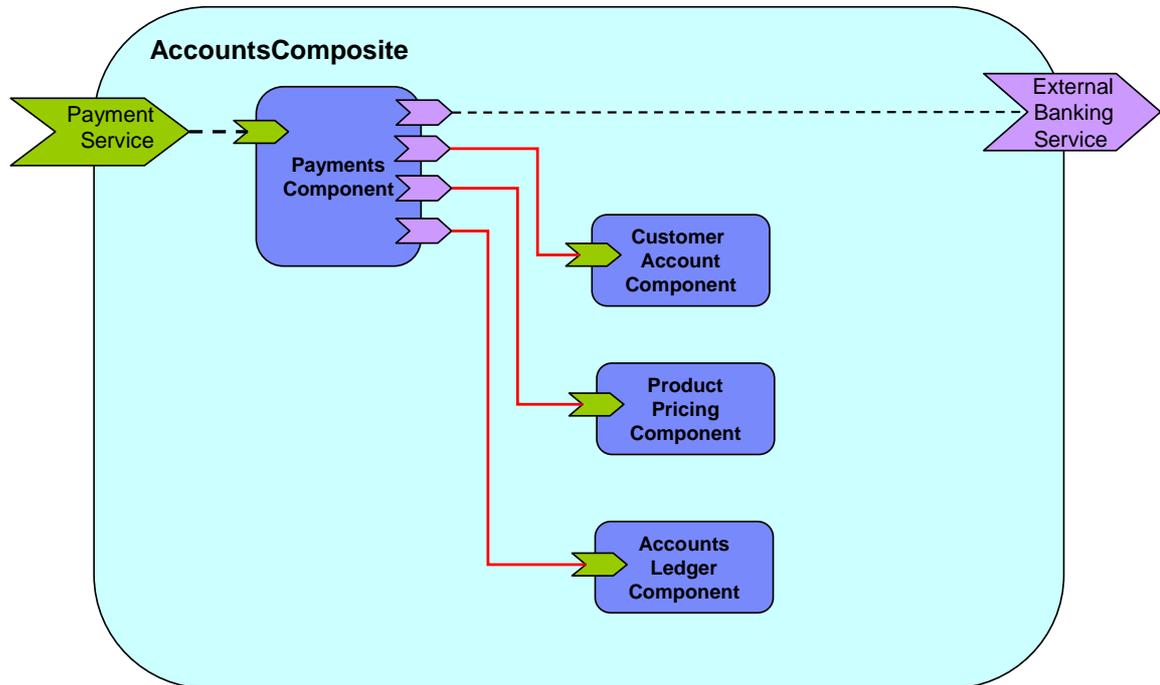


Figure 12: Example Composite for Autowire

First, the composite using explicit wires:

```

1947 <?xml version="1.0" encoding="UTF-8"?>
1948 <!-- Autowire Example - No autowire -->
1949 <composite xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
1950 xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1951 xmlns:foo="http://foo.com"
1952 targetNamespace="http://foo.com"
1953 name="AccountComposite">
1954
1955 <service name="PaymentService" promote="PaymentsComponent"/>
1956
1957 <component name="PaymentsComponent">
1958 <implementation.java class="com.foo.accounts.Payments"/>
1959 <service name="PaymentService"/>
1960 <reference name="CustomerAccountService"
1961 target="CustomerAccountComponent"/>
1962 <reference name="ProductPricingService"
1963 target="ProductPricingComponent"/>
1964 <reference name="AccountsLedgerService"
1965 target="AccountsLedgerComponent"/>
1966 <reference name="ExternalBankingService"/>
1967 </component>
1968
1969 <component name="CustomerAccountComponent">
1970 <implementation.java class="com.foo.accounts.CustomerAccount"/>
1971 </component>
1972
1973 <component name="ProductPricingComponent">
1974 <implementation.java class="com.foo.accounts.ProductPricing"/>
1975 </component>
1976
1977 <component name="AccountsLedgerComponent">

```

```

1978 <implementation.composite name="foo:AccountsLedgerComposite"/>
1979 </component>
1980
1981 <reference name="ExternalBankingService"
1982 promote="PaymentsComponent/ExternalBankingService"/>
1983
1984 </composite>
1985

```

1986 Secondly, the composite using autowire:

```

1987 <?xml version="1.0" encoding="UTF-8"?>
1988 <!-- Autowire Example - With autowire -->
1989 <composite xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
1990 xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
1991 xmlns:foo="http://foo.com"
1992 targetNamespace="http://foo.com"
1993 name="AccountComposite">
1994
1995 <service name="PaymentService" promote="PaymentsComponent">
1996 <interface.java class="com.foo.PaymentServiceInterface"/>
1997 </service>
1998
1999 <component name="PaymentsComponent" autowire="true">
2000 <implementation.java class="com.foo.accounts.Payments"/>
2001 <service name="PaymentService"/>
2002 <reference name="CustomerAccountService"/>
2003 <reference name="ProductPricingService"/>
2004 <reference name="AccountsLedgerService"/>
2005 <reference name="ExternalBankingService"/>
2006 </component>
2007
2008 <component name="CustomerAccountComponent">
2009 <implementation.java class="com.foo.accounts.CustomerAccount"/>
2010 </component>
2011
2012 <component name="ProductPricingComponent">
2013 <implementation.java class="com.foo.accounts.ProductPricing"/>
2014 </component>
2015
2016 <component name="AccountsLedgerComponent">
2017 <implementation.composite name="foo:AccountsLedgerComposite"/>
2018 </component>
2019
2020 <reference name="ExternalBankingService"
2021 promote="PaymentsComponent/ExternalBankingService"/>
2022
2023 </composite>

```

2024 In this second case, autowire is set on for the PaymentsComponent and there are no explicit wires  
2025 for any of its references – the wires are created automatically through autowire.

2026 **Note:** In the second example, it would be possible to omit all of the service and reference  
2027 elements from the PaymentsComponent. They are left in for clarity, but if they are omitted, the  
2028 component service and references still exist, since they are provided by the implementation used  
2029 by the component.

2030

## 2031 6.5 Using Composites as Component Implementations

2032 Composites may form **component implementations** in higher-level composites – in other words  
2033 the higher-level composites can have components which are implemented by composites.

2034 When a composite is used as a component implementation, it defines a boundary of visibility.  
2035 Components within the composite cannot be referenced directly by the using component. The  
2036 using component can only connect wires to the services and references of the used composite and  
2037 set values for any properties of the composite. The internal construction of the composite is  
2038 invisible to the using component. The boundary of visibility, sometimes called encapsulation, can  
2039 be enforced when assembling components and composites, but such encapsulation structures  
2040 might not be enforceable in a particular implementation language.

2041 A composite used as a component implementation must also honor a completeness contract. The  
2042 services, references and properties of the composite form a contract (represented by the  
2043 component type of the composite) which is relied upon by the using component. The concept of  
2044 completeness of the composite implies that, once all <include/> element processing is performed  
2045 on the composite:

- 2046 1. For a composite used as a component implementation, each composite service offered by  
2047 the composite MUST promote a component service of a component that is within the  
2048 composite. [ASM60032]
- 2049 2. For a composite used as a component implementation, every component reference of  
2050 components within the composite with a multiplicity of 1..1 or 1..n MUST be wired or  
2051 promoted (according to the various rules for specifying target services for a component  
2052 reference described in section 5.3.1). [ASM60033]
- 2053 3. For a composite used as a component implementation, all properties of components within  
2054 the composite, where the underlying component implementation specifies  
2055 "mustSupply=true" for the property, MUST either specify a value for the property or  
2056 source the value from a composite property. [ASM60034]

2057 The component type of a composite is defined by the set of composite service elements,  
2058 composite reference elements and composite property elements that are the children of the  
2059 composite element.

2060 Composites are used as component implementations through the use of the  
2061 **implementation.composite** element as a child element of the component. The schema snippet  
2062 for the implementation.composite element is:

2063

```
2064 <!-- implementation.composite pseudo-schema -->
2065 <implementation.composite name="xs:QName" requires="list of xs:QName"?
2066 policySets="list of xs:QName"?>
```

2067

2068 The implementation.composite element has the following attributes:

- 2069 • **name (1..1)** – the name of the composite used as an implementation. The @name  
2070 attribute of an <implementation.composite/> element MUST contain the QName of a  
2071 composite in the SCA Domain. [ASM60030]
- 2072 • **requires : QName (0..n)** – a list of required policy intents. See the [Policy Framework  
2073 specification \[10\]](#) for a description of this attribute. Specified **required intents** add to or  
2074 further qualify the required intents defined for the promoted component reference.
- 2075 • **policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification  
2076 \[10\]](#) for a description of this attribute.

2077

### 2078 6.5.1 Example of Composite used as a Component Implementation

2079

2080 The following is an example of a composite which contains two components, each of which is  
2081 implemented by a composite:  
2082

```
2083 <?xml version="1.0" encoding="UTF-8"?>
2084 <!-- CompositeComponent example -->
2085 <composite xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
2086 xsd:schemaLocation="http://docs.oasis-open.org/ns/opencsa/sca/200712
2087 file:/C:/Strategy/SCA/v09_osoaschemas/schemas/sca.xsd"
2088 xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2089 targetNamespace="http://foo.com"
2090 xmlns:foo="http://foo.com"
2091 name="AccountComposite">
2092
2093 <service name="AccountService" promote="AccountServiceComponent">
2094 <interface.java interface="services.account.AccountService"/>
2095 <binding.ws port="AccountService#
2096 wsdl.endpoint(AccountService/AccountServiceSOAP)"/>
2097 </service>
2098
2099 <reference name="stockQuoteService"
2100 promote="AccountServiceComponent/StockQuoteService">
2101 <interface.java
2102 interface="services.stockquote.StockQuoteService"/>
2103 <binding.ws
2104 port="http://www.quickstockquote.com/StockQuoteService#
2105 wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
2106 </reference>
2107
2108 <property name="currency" type="xsd:string">EURO</property>
2109
2110 <component name="AccountServiceComponent">
2111 <implementation.composite name="foo:AccountServiceCompositel"/>
2112
2113 <reference name="AccountDataService" target="AccountDataService"/>
2114 <reference name="StockQuoteService"/>
2115
2116 <property name="currency" source="$currency"/>
2117 </component>
2118
2119 <component name="AccountDataService">
2120 <implementation.composite name="foo:AccountDataServiceComposite"/>
2121
2122 <property name="currency" source="$currency"/>
2123 </component>
2124
2125 </composite>
2126
```

## 2127 6.6 Using Composites through Inclusion

2128 In order to assist team development, composites may be developed in the form of multiple  
2129 physical artifacts that are merged into a single logical unit.

2130 A composite may include another composite by using the **include** element. This provides a  
2131 recursive inclusion capability. The semantics of included composites are that the element content  
2132 children of the included composite are inlined, with certain modification, into the using composite.  
2133 This is done recursively till the resulting composite does not contain an **include** element. The

2134 outer included composite element itself is discarded in this process – only its contents are included  
2135 as described below:

- 2136 1. All the element content children of the included composite are inlined in the including  
2137 composite.
- 2138 2. The attributes **targetNamespace**, **name**, **constrainingType**, and **local** of the included  
2139 composites are discarded.
- 2140 3. All the namespace declaration on the included composite element are added to the inlined  
2141 element content children unless the namespace binding is overridden by the element  
2142 content children.
- 2143 4. The attribute **autowire**, if specified on the included composite, is included on all inlined  
2144 component element children unless the component child already specifies that attribute.
- 2145 5. The attribute values of **requires** and **policySet**, if specified on the included composite,  
2146 are merged with corresponding attribute on the inlined component, service and reference  
2147 children elements. Merge in this context means a set union.
- 2148 6. Extension attributes ,if present on the included composite, must follow the rules defined  
2149 for that extension. Authors of attribute extensions on the composite element must define  
2150 rules for inclusion.

2151 If the included composite has the value *true* for the attribute **local** then the including composite  
2152 must have the same value for the **local** attribute, else it is considered an error.

2153 The composite file used for inclusion can have any contents, but its document root element must  
2154 be **composite**. The composite element may contain any of the elements which are valid as child  
2155 elements of a composite element, namely components, services, references, wires and includes.  
2156 There is no need for the content of an included composite to be complete, so that artifacts defined  
2157 within the using composite or in another associated included composite file may be referenced. For  
2158 example, it is permissible to have two components in one composite file while a wire specifying  
2159 one component as the source and the other as the target can be defined in a second included  
2160 composite file.

2161 The SCA runtime MUST raise an error if the composite resulting from the inclusion of one  
2162 composite into another is invalid. [ASM60031] For example, it is an error if there are duplicated  
2163 elements in the using composite (eg. two services with the same uri contributed by different  
2164 included composites). It is not considered an error if the (using) composite resulting from the  
2165 inclusion is incomplete (eg. wires with non-existent source or target). Such incomplete resulting  
2166 composites are permitted to allow recursive composition.

2167 The following snippet shows the pseudo-schema for the include element.

2168

```
2169 <?xml version="1.0" encoding="UTF-8"?>
2170 <!-- Include snippet -->
2171 <composite ...>
2172 ...
2173 <include name="xs:QName"/> *
2174 ...
2175 </composite>
2176
```

2177 The include element has the following **attribute**:

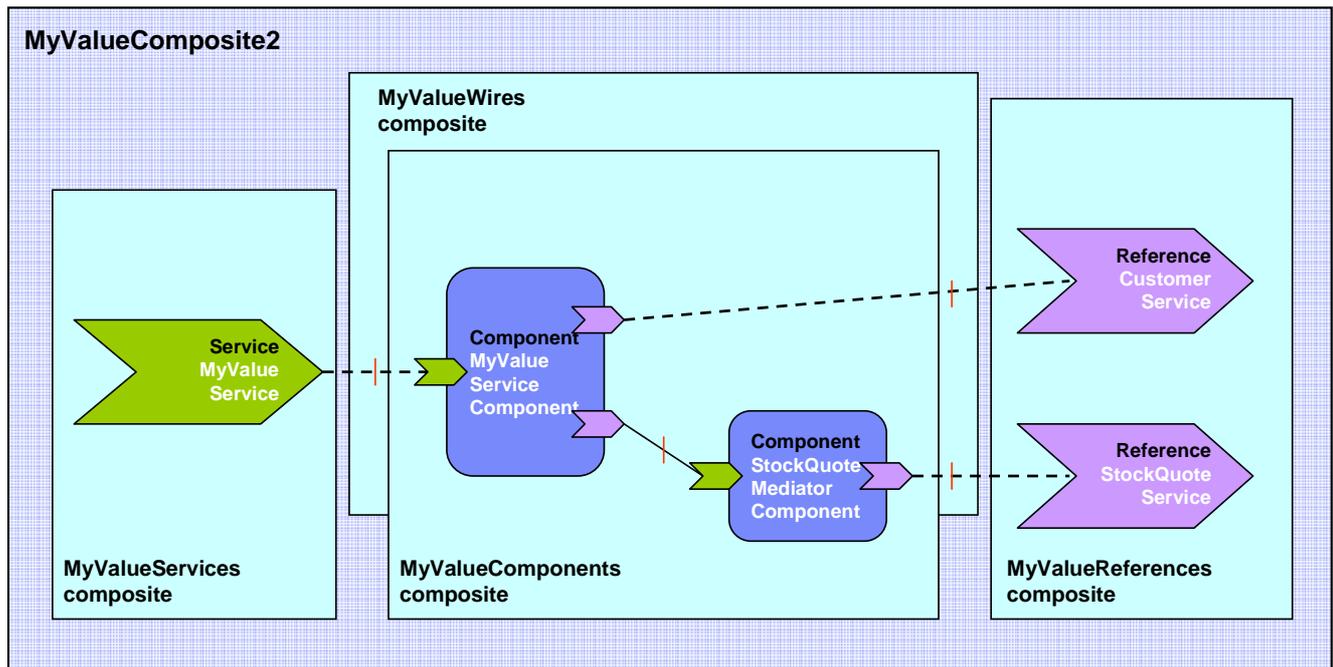
- 2178 • **name (required)** – the name of the composite that is included.

2179

## 2180 6.6.1 Included Composite Examples

2181

2182 The following figure shows the assembly diagram for the MyValueComposite2 containing four  
 2183 included composites. The **MyValueServices composite** contains the MyValueService service. The  
 2184 **MyValueComponents composite** contains the MyValueServiceComponent and the  
 2185 StockQuoteMediatorComponent as well as the wire between them. The **MyValueReferences**  
 2186 **composite** contains the CustomerService and StockQuoteService references. The **MyValueWires**  
 2187 **composite** contains the wires that connect the MyValueService service to the  
 2188 MyValueServiceComponent, that connect the customerService reference of the  
 2189 MyValueServiceComponent to the CustomerService reference, and that connect the  
 2190 stockQuoteService reference of the StockQuoteMediatorComponent to the StockQuoteService  
 2191 reference. Note that this is just one possible way of building the MyValueComposite2 from a set of  
 2192 included composites.



2193  
 2194  
 2195 *Figure 13 MyValueComposite2 built from 4 included composites*

2196  
 2197 The following snippet shows the contents of the MyValueComposite2.composite file for the  
 2198 MyValueComposite2 built using included composites. In this sample it only provides the name of  
 2199 the composite. The composite file itself could be used in a scenario using included composites to  
 2200 define components, services, references and wires.

```
2201
2202 <?xml version="1.0" encoding="ASCII"?>
2203 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2204 targetNamespace="http://foo.com"
2205 xmlns:foo="http://foo.com"
2206 name="MyValueComposite2" >
2207
2208 <include name="foo:MyValueServices"/>
2209 <include name="foo:MyValueComponents"/>
2210 <include name="foo:MyValueReferences"/>
2211 <include name="foo:MyValueWires"/>
2212
2213 </composite>
2214
```

2215 The following snippet shows the content of the MyValueServices.composite file.

```
2216
2217 <?xml version="1.0" encoding="ASCII"?>
2218 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2219 targetNamespace="http://foo.com"
2220 xmlns:foo="http://foo.com"
2221 name="MyValueServices" >
2222
2223 <service name="MyValueService" promote="MyValueServiceComponent">
2224 <interface.java interface="services.myvalue.MyValueService"/>
2225 <binding.ws port="http://www.myvalue.org/MyValueService#
2226 wsdl.endpoint(MyValueService/MyValueServiceSOAP)"/>
2227 </service>
2228
2229 </composite>
2230
```

2231 The following snippet shows the content of the MyValueComponents.composite file.

```
2232
2233 <?xml version="1.0" encoding="ASCII"?>
2234 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2235 targetNamespace="http://foo.com"
2236 xmlns:foo="http://foo.com"
2237 name="MyValueComponents" >
2238
2239 <component name="MyValueServiceComponent">
2240 <implementation.java
2241 class="services.myvalue.MyValueServiceImpl"/>
2242 <property name="currency">EURO</property>
2243 </component>
2244
2245 <component name="StockQuoteMediatorComponent">
2246 <implementation.java class="services.myvalue.SQMediatorImpl"/>
2247 <property name="currency">EURO</property>
2248 </component>
2249
2250 <composite>
2251
```

2252 The following snippet shows the content of the MyValueReferences.composite file.

```
2253
2254 <?xml version="1.0" encoding="ASCII"?>
2255 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2256 targetNamespace="http://foo.com"
2257 xmlns:foo="http://foo.com"
2258 name="MyValueReferences" >
2259
2260 <reference name="CustomerService"
2261 promote="MyValueServiceComponent/CustomerService">
2262 <interface.java interface="services.customer.CustomerService"/>
2263 <binding.sca/>
2264 </reference>
2265
2266 <reference name="StockQuoteService"
2267 promote="StockQuoteMediatorComponent">
2268 <interface.java
```

```

2269 interface="services.stockquote.StockQuoteService" />
2270 <binding.ws port="http://www.stockquote.org/StockQuoteService#
2271 wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)" />
2272 </reference>
2273
2274 </composite>
2275

```

2276 The following snippet shows the content of the MyValueWires.composite file.

```

2277
2278 <?xml version="1.0" encoding="ASCII"?>
2279 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2280 targetNamespace="http://foo.com"
2281 xmlns:foo="http://foo.com"
2282 name="MyValueWires" >
2283
2284 <wire source="MyValueServiceComponent/stockQuoteService"
2285 target="StockQuoteMediatorComponent" />
2286
2287 </composite>

```

## 2288 6.7 Composites which Include Component Implementations of 2289 Multiple Types

2290 A Composite containing multiple components can have multiple component implementation types.  
2291 For example, a Composite may include one component with a Java POJO as its implementation  
2292 and another component with a BPEL process as its implementation.

## 2293 6.8 Structural URI of Components

2294 The **structural URI** is a relative URI that describes each use of a given component in the Domain,  
2295 relative to the URI of the domain itself. It is never specified explicitly, but it calculated from the  
2296 configuration of the components configured into the Domain.

2297 A component in a composite may be used more than once in the domain, if its containing  
2298 composite is used as the implementation of more than one higher-level component. The structural  
2299 URI may be used to separately identify each use of a component - for example, the structural URI  
2300 may be used to attach different policies to each separate use of a component.

2301 For components directly deployed into the domain, the structural URI is simply the name of the  
2302 component.

2303 Where components are nested within a composite which is used as the implementation of a higher  
2304 level component, the structural URI consists of the name of the nested component prepended with  
2305 each of the names of the components upto and including the domain level component.

2306 For example, consider a component named Component1 at the domain level, where its  
2307 implementation is Composite1 which in turn contains a component named Component2, which is  
2308 implemented by Composite2 which contains a component named Component3. The three  
2309 components in this example have the following structural URIs:

- 2310 1. Component1: Component1
- 2311 2. Component2: Component1/Component2
- 2312 3. Component3: Component1/Component2/Component3

2313 The structural URI can also be extended to refer to specific parts of a component, such as a  
2314 service or a reference, by appending an appropriate fragment identifier to the component's  
2315 structural URI, as follows:

2316  
2317  
2318

- Service:  
#service(servicename)

2319  
2320  
2321

- Reference:  
#reference(referencename)

2322  
2323  
2324

- Service binding:  
#service-binding(servicename/bindingname)

2325  
2326

- Reference binding:  
#reference-binding(referencename/bindingname)

2327  
2328  
2329

So, for example, the structural URI of the service named "testservice" of component "Component1" is Component1#service(testservice).

---

## 2330 7 ConstrainingType

2331 SCA allows a component, and its associated implementation, to be constrained by a  
2332 **constrainingType**. The `constrainingType` element provides assistance in developing top-down  
2333 usecases in SCA, where an architect or assembler can define the structure of a composite,  
2334 including the required form of component implementations, before any of the implementations are  
2335 developed.

2336 A `constrainingType` is expressed as an element which has services, reference and properties as  
2337 child elements and which can have intents applied to it. The `constrainingType` is independent of  
2338 any implementation. Since it is independent of an implementation it cannot contain any  
2339 implementation-specific configuration information or defaults. Specifically, it cannot contain  
2340 bindings, `policySets`, property values or default wiring information. The `constrainingType` is  
2341 applied to a component through a `constrainingType` attribute on the component.

2342 A `constrainingType` provides the "shape" for a component and its implementation. Any component  
2343 configuration that points to a `constrainingType` is constrained by this shape. The `constrainingType`  
2344 specifies the services, references and properties that **MUST** be implemented by the  
2345 implementation of the component to which the `constrainingType` is attached. [ASM70001] This  
2346 provides the ability for the implementer to program to a specific set of services, references and  
2347 properties as defined by the `constrainingType`. Components are therefore configured instances of  
2348 implementations and are constrained by an associated `constrainingType`.

2349 If the configuration of the component or its implementation do not conform to the  
2350 `constrainingType` specified on the component element, the SCA runtime **MUST** raise an error.  
2351 [ASM70002]

2352 A `constrainingType` is represented by a **constrainingType** element. The following snippet shows  
2353 the pseudo-schema for the composite element.

2354

```
2355 <?xml version="1.0" encoding="ASCII"?>
2356 <!-- ConstrainingType schema snippet -->
2357 <constrainingType xmlns="http://docs.oasis-
2358 open.org/ns/opencsa/sca/200712"
2359 targetNamespace="xs:anyURI"?
2360 name="xs:NCName" requires="list of xs:QName"?>
2361
2362 <service name="xs:NCName" requires="list of xs:QName"?>*
2363 <interface ... />?
2364 </service>
2365
2366 <reference name="xs:NCName"
2367 multiplicity="0..1 or 1..1 or 0..n or 1..n"?
2368 requires="list of xs:QName"?>*
2369 <interface ... />?
2370 </reference>
2371
2372 <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
2373 many="xs:boolean"? mustSupply="xs:boolean"?>*
2374 default-property-value?
2375 </property>
2376 </constrainingType>
```

2380 The `constrainingType` element has the following **attributes**:

- 2381 • **name (1..1)** – the name of the constrainingType. The form of a constrainingType name is  
2382 an XML QName, in the namespace identified by the targetNamespace attribute. The name  
2383 attribute of the constraining type MUST be unique in the SCA domain. [ASM70003]
- 2384 • **targetNamespace (0..1)** – an identifier for a target namespace into which the  
2385 constrainingType is declared
- 2386 • **requires (0..1)** – a list of policy intents. See [the Policy Framework specification \[10\]](#) for  
2387 a description of this attribute.

2388 ConstrainingType contains **zero or more properties, services, references**.

2389

2390 When an implementation is constrained by a constrainingType its component type MUST contain  
2391 all the services, references and properties specified in the constrainingType. [ASM70004] The  
2392 constraining type's references and services will have interfaces specified and can have intents  
2393 specified. An implementation MAY contain additional services, additional optional references  
2394 (multiplicity 0..1 or 0..n) and additional optional properties beyond those declared in the  
2395 constraining type, but MUST NOT contain additional non-optional references (multiplicity 1..1 or  
2396 1..n) or additional non-optional properties (a property with mustSupply=true). [ASM70005]

2397 When a component is constrained by a constrainingType via the "constrainingType" attribute, the  
2398 entire componentType associated with the component and its implementation is not visible to the  
2399 containing composite. The containing composite can only see a projection of the componentType  
2400 associated with the component and implementation as scoped by the constrainingType of the  
2401 component. Additional services, references and properties provided by the implementation which  
2402 are not declared in the constrainingType associated with a component MUST NOT be configured in  
2403 any way by the containing composite. [ASM70006] This requirement ensures that the  
2404 constrainingType contract cannot be violated by the composite.

2405 The constrainingType can include required intents on any element. Those intents are applied to  
2406 any component that uses that constrainingType. In other words, if requires="reliability" exists on  
2407 a constrainingType, or its child service or reference elements, then a constrained component or its  
2408 implementation must include requires="reliability" on the component or implementation or on its  
2409 corresponding service or reference. A component or implementation can use a qualified form of  
2410 an intent specified in unqualified form in the constrainingType, but if the constrainingType uses  
2411 the qualified form of an intent, then the component or implementation MUST also use the qualified  
2412 form, otherwise there is an error. [ASM70007]

2413 A constrainingType can be applied to an implementation. In this case, the implementation's  
2414 componentType has a constrainingType attribute set to the QName of the constrainingType.

2415

## 2416 7.1 Example constrainingType

2417 The following snippet shows the contents of the component called "MyValueServiceComponent"  
2418 which is constrained by the constrainingType myns:CT. The componentType associated with the  
2419 implementation is also shown.

2420

```
2421 <component name="MyValueServiceComponent" constrainingType="myns:CT">
2422 <implementation.java class="services.myvalue.MyValueServiceImpl"/>
2423 <property name="currency">EURO</property>
2424 <reference name="customerService" target="CustomerService">
2425 <binding.ws ...>
2426 <reference name="StockQuoteService"
2427 target="StockQuoteMediatorComponent"/>
2428 </component>
2429
2430 <constrainingType name="CT"
2431 targetNamespace="http://myns.com">
```

```
2432 <service name="MyValueService">
2433 <interface.java interface="services.myvalue.MyValueService"/>
2434 </service>
2435 <reference name="customerService">
2436 <interface.java interface="services.customer.CustomerService"/>
2437 </reference>
2438 <reference name="stockQuoteService">
2439 <interface.java interface="services.stockquote.StockQuoteService"/>
2440 </reference>
2441 <property name="currency" type="xsd:string"/>
2442 </constrainingType>
2443
```

2444 The component MyValueServiceComponent is constrained by the constrainingType CT which  
2445 means that it must provide:

- 2446 • service **MyValueService** with the interface services.myvalue.MyValueService
- 2447 • reference **customerService** with the interface services.stockquote.StockQuoteService
- 2448 • reference **stockQuoteService** with the interface services.stockquote.StockQuoteService
- 2449 • property **currency** of type xsd:string.

## 2450 8 Interface

2451 **Interfaces** define one or more business functions. These business functions are provided by  
2452 Services and are used by References. A Service offers the business functionality of exactly one  
2453 interface for use by other components. Each interface defines one or more service **operations**  
2454 and each operation has zero or one **request (input) message** and zero or one **response**  
2455 **(output) message**. The request and response messages can be simple types such as a string  
2456 value or they can be complex types.

2457 SCA currently supports the following interface type systems:

- 2458 • Java interfaces
- 2459 • WSDL 1.1 portTypes ([Web Services Definition Language \[8\]](#))
- 2460 • C++ classes
- 2461 • Collections of 'C' functions

2462 SCA is also extensible in terms of interface types. Support for other interface type systems can be  
2463 added through the extensibility mechanisms of SCA, as described in [the Extension Model section](#).

2464

2465 The following snippet shows the definition for the **interface** base element.

2466

```
2467 <interface requires="list of xs:QName"? policySets="list of xs:QName"?/>
```

2468

2469 The **interface** base element has the following **attributes**:

- 2470 • **requires : QName (0..n)** – a list of policy intents. See the [Policy Framework specification](#)  
2471 [\[10\]](#) for a description of this attribute
- 2472 • **policySets : QName (0..n)** – a list of policy sets. See the [Policy Framework specification](#)  
2473 [\[10\]](#) for a description of this attribute.

2474

2475 For information about Java interfaces, including details of SCA-specific annotations, see the SCA  
2476 Java Common Annotations and APIs specification [SCA-Common-Java].

2477 For information about WSDL interfaces, including details of SCA-specific extensions, see SCA-  
2478 Specific Aspects for WSDL Interfaces and WSDL Interface Type.

2479 For information about C++ interfaces, see the SCA C++ Client and Implementation Model  
2480 specification [SCA-CPP-Client].

2481 For information about C interfaces, see the SCA C Client and Implementation Model specification  
2482 [SCA-C-Client].

### 2483 8.1 Local and Remotable Interfaces

2484 A remotable service is one which may be called by a client which is running in an operating system  
2485 process different from that of the service itself (this also applies to clients running on different  
2486 machines from the service). Whether a service of a component implementation is remotable is  
2487 defined by the interface of the service. WSDL defined interfaces are always remotable. See the  
2488 relevant specifications for details of interfaces defined using other languages.

2489

2490 The style of remotable interfaces is typically **coarse grained** and intended for **loosely coupled**  
2491 interactions. Remotable service Interfaces MUST NOT make use of **method or operation**  
2492 **overloading**. [\[ASM80002\]](#) This restriction on operation overloading for remotable services aligns

2493 with the WSDL 2.0 specification, which disallows operation overloading, and also with the WS-I  
2494 Basic Profile 1.1 (section 4.5.3 - R2304) which has a constraint which disallows operation  
2495 overloading when using WSDL 1.1.

2496  
2497 Independent of whether the remotable service is called remotely from outside the process where  
2498 the service runs or from another component running in the same process, the data exchange  
2499 semantics are **by-value**.

2500 Implementations of remotable services can modify input messages (parameters) during or after  
2501 an invocation and can modify return messages (results) after the invocation. If a remotable  
2502 service is called locally or remotely, the SCA container MUST ensure sure that no modification of  
2503 input messages by the service or post-invocation modifications to return messages are seen by  
2504 the caller. [ASM80003]

2505 Here is a snippet which shows an example of a remotable java interface:

```
2506
2507 package services.hello;
2508
2509 @Remotable
2510 public interface HelloService {
2511 String hello(String message);
2512 }
2513
2514
```

2515 It is possible for the implementation of a remotable service to indicate that it can be called using  
2516 by-reference data exchange semantics when it is called from a component in the same process.  
2517 This can be used to improve performance for service invocations between components that run in  
2518 the same process. This can be done using the @AllowsPassByReference annotation (see the [Java  
2519 Client and Implementation Specification](#)).

2520 A service typed by a local interface can only be called by clients that are running in the same  
2521 process as the component that implements the local service. Local services cannot be published  
2522 via remotable services of a containing composite. In the case of Java a local service is defined by a  
2523 Java interface definition without a **@Remotable** annotation.

2524 The style of local interfaces is typically **fine grained** and intended for **tightly coupled**  
2525 interactions. Local service interfaces can make use of **method or operation overloading**.

2526 The data exchange semantic for calls to services typed by local interfaces is **by-reference**.

2527

## 2528 8.2 Bidirectional Interfaces

2529 The relationship of a business service to another business service is often peer-to-peer, requiring  
2530 a two-way dependency at the service level. In other words, a business service represents both a  
2531 consumer of a service provided by a partner business service and a provider of a service to the  
2532 partner business service. This is especially the case when the interactions are based on  
2533 asynchronous messaging rather than on remote procedure calls. The notion of **bidirectional  
2534 interfaces** is used in SCA to directly model peer-to-peer bidirectional business service  
2535 relationships.

2536 An interface element for a particular interface type system needs to allow the specification of an  
2537 optional callback interface. If a callback interface is specified, SCA refers to the interface as a  
2538 whole as a bidirectional interface.

2539 The following snippet shows the interface element defined using Java interfaces with an optional  
2540 callbackInterface attribute.

2541

```
2542 <interface.java interface="services.invoicing.ComputePrice"
2543 callbackInterface="services.invoicing.InvoiceCallback"/>
```

2544

2545 If a service is defined using a bidirectional interface element then its implementation implements  
2546 the interface, and its implementation uses the callback interface to converse with the client that  
2547 called the service interface.

2548

2549 If a reference is defined using a bidirectional interface element, the client component  
2550 implementation using the reference calls the referenced service using the interface. The client  
2551 MUST provide an implementation of the callback interface. [ASM80004]

2552 Callbacks can be used for both remotable and local services. Either both interfaces of a  
2553 bidirectional service MUST be remotable, or both MUST be local. A bidirectional service MUST NOT  
2554 mix local and remote services. [ASM80005]

2555 Note that an interface document such as a WSDL file or a Java interface can contain annotations  
2556 that declare a callback interface for a particular interface (see [the section on WSDL Interface type](#)  
2557 and the Java Common Annotations and APIs specification [SCA-Common-Java]). Whenever an  
2558 interface document declaring a callback interface is used in the declaration of an <interface/>  
2559 element in SCA, it MUST be treated as being bidirectional with the declared callback interface.  
2560 [ASM80010] In such cases, there is no requirement for the <interface/> element to declare the  
2561 callback interface explicitly.

2562 If an <interface/> element references an interface document which declares a callback interface  
2563 and also itself contains a declaration of a callback interface, the two callback interfaces MUST be  
2564 compatible. [ASM80011]

2565 Where a component uses an implementation and the component configuration explicitly declares  
2566 an interface for a service or a reference, if the matching service or reference declaration in the  
2567 component type declares an interface which has a callback interface, then the component interface  
2568 declaration MUST also declare a compatible interface with a compatible callback interface.  
2569 [ASM80012] If the service or reference declaration in the component type declares an interface  
2570 without a callback interface, then the component configuration for the corresponding service or  
2571 reference MUST NOT declare an interface with a callback interface. [ASM80013]

2572 Where a composite declares an interface for a composite service or a composite reference, if the  
2573 promoted service or promoted reference has an interface which has a callback interface, then the  
2574 interface declaration for the composite service or the composite reference MUST also declare a  
2575 compatible interface with a compatible callback interface. [ASM80014] If the promoted service or  
2576 promoted reference has an interface without a callback interface, then the interface declaration for  
2577 the composite service or composite reference MUST NOT declare a callback interface.  
2578 [ASM80015]

2579 See Section 6.4 Wires for a definition of "compatible interfaces".

2580 In a bidirectional interface, the service interface can have more than one operation defined, and  
2581 the callback interface can also have more than one operation defined. SCA runtimes MUST allow  
2582 an invocation of any operation on the service interface to be followed by zero, one or many  
2583 invocations of any of the operations on the callback interface. [ASM80009] These callback  
2584 operations can be invoked either before or after the operation on the service interface has  
2585 returned a response message, if there is one.

2586 For a given invocation of a service operation, which operations are invoked on the callback  
2587 interface, when these are invoked, the number of operations invoked, and their sequence are not  
2588 described by SCA. It is possible that this metadata about the bidirectional interface can be  
2589 supplied through mechanisms outside SCA. For example, it might be provided as a written  
2590 description attached to the callback interface.

### 2591 **8.3 Conversational Interfaces**

2592 Services sometimes cannot easily be defined so that each operation stands alone and is  
2593 completely independent of the other operations of the same service. Instead, there is a sequence  
2594 of operations that must be called in order to achieve some higher level goal. SCA calls this

2595 sequence of operations a **conversation**. If the service uses a bidirectional interface, the  
2596 conversation may include both operations and callbacks.

2597 Such **conversational services** are typically managed by using conversation identifiers that are  
2598 either (1) part of the application data (message parts or operation parameters) or 2)  
2599 communicated separately from application data (possibly in headers). SCA introduces the concept  
2600 of **conversational interfaces** for describing the interface contract for conversational services of  
2601 the second form above. With this form, it is possible for the runtime to automatically manage the  
2602 conversation, with the help of an appropriate binding specified at deployment. SCA does not  
2603 standardize any aspect of conversational services that are maintained using application data.  
2604 Such services are neither helped nor hindered by SCA's conversational service support.

2605 Conversational services typically involve state data that relates to the conversation that is taking  
2606 place. The creation and management of the state data for a conversation has a significant impact  
2607 on the development of both clients and implementations of conversational services.

2608

2609 Traditionally, application developers who have needed to write conversational services have been  
2610 required to write a lot of plumbing code. They need to:

2611

- 2612 - choose or define a protocol to communicate conversational (correlation) information  
2613 between the client & provider
- 2614 - route conversational messages in the provider to a machine that can handle that  
2615 conversation, while handling concurrent data access issues
- 2616 - write code in the client to use/encode the conversational information
- 2617 - maintain state that is specific to the conversation, sometimes persistently and  
2618 transactionally, both in the implementation and the client.

2619

2620 SCA makes it possible to divide the effort associated with conversational services between a  
2621 number of roles:

- 2622 - Application Developer: Declares that a service interface is conversational (leaving the  
2623 details of the protocol up to the binding). Uses lifecycle semantics, APIs or other  
2624 programmatic mechanisms (as defined by the implementation-type being used) to  
2625 manage conversational state.
- 2626 - Application Assembler: chooses a binding that can support conversations
- 2627 - Binding Provider: implements a protocol that can pass conversational information with  
2628 each operation request/response.
- 2629 - Implementation-Type Provider: defines APIs and/or other programmatic mechanisms for  
2630 application developers to access conversational information. Optionally implements  
2631 instance lifecycle semantics that automatically manage implementation state based on  
2632 the binding's conversational information.

2633

2634 There is a policy intent with the name **conversational** which is used to mark an interface as being  
2635 conversational in nature. Where a service or a reference has a conversational interface, the  
2636 conversational intent MUST be attached either to the interface itself, or to the service or reference  
2637 using the interface. [ASM80006] How to attach the conversational intent to an interface depends  
2638 on the type of the interface. For a WSDL interface, this is described in section 8.4 "[SCA-Specific  
2639 Aspects for WSDL Interfaces](#)". For a Java interface, it is described in the Java Common  
2640 Annotations and APIs specification. Note that setting the conversational intent on the service or  
2641 reference element is useful when reusing an existing interface definition that contains no SCA  
2642 information, since it requires no modification of the interface artifact.

2643 The meaning of the conversational intent is that both the client and the provider of the interface  
2644 can assume that messages (in either direction) will be handled as part of an ongoing conversation

2645 without depending on identifying information in the body of the message (i.e. in parameters of the  
2646 operations). In effect, the conversation interface specifies a high-level abstract protocol that must  
2647 be satisfied by any actual binding/policy combination used by the service.

2648 Examples of binding/policy combinations that support conversational interfaces are:

- 2649 - Web service binding with a WS-RM policy
- 2650 - Web service binding with a WS-Addressing policy
- 2651 - Web service binding with a WS-Context policy
- 2652 - JMS binding with a conversation policy that uses the JMS correlationID header

2653  
2654 Conversations occur between one client and one target service. Consequently, requests originating  
2655 from one client to multiple target conversational services will result in multiple conversations. For  
2656 example, if a client A calls services B and C, both of which implement conversational interfaces,  
2657 two conversations result, one between A and B and another between A and C. Likewise, requests  
2658 flowing through multiple implementation instances will result in multiple conversations. For  
2659 example, a request flowing from A to B and then from B to C will involve two conversations (A and  
2660 B, B and C). In the previous example, if a request was then made from C to A, a third  
2661 conversation would result (and the implementation instance for A would be different from the one  
2662 making the original request).

2663 Invocation of any operation of a conversational interface can start a conversation. The decision on  
2664 whether an operation starts a conversation depends on the component's implementation and its  
2665 implementation type. Implementation types can support components which provide conversational  
2666 services. If an implementation type does provide this support, the specification for that  
2667 implementation type defines a mechanism for determining when a new conversation should be  
2668 used for an operation (for example, in Java, the conversation is new on the first use of an injected  
2669 reference; in BPEL, the conversation is new when the client's partnerLink comes into scope).

2670  
2671 One or more operations in a conversational interface can be annotated with an  
2672 **endsConversation** annotation (the mechanism for annotating the interface depends on the  
2673 interface type) which indicates that when the operation is invoked, the conversation is at an end.  
2674 Where an interface is **bidirectional**, operations may also be annotated in this way on operations  
2675 of the callback interface. When a conversation ending operation is called, it indicates to both the  
2676 client and the service provider that the conversation is complete. Once an operation marked with  
2677 endsConversation has been invoked, any subsequent attempts to call an operation or a callback  
2678 operation associated with the same conversation MUST generate a sca:ConversationViolation fault.  
2679 [\[ASM80007\]](#)

2680 A sca:ConversationViolation fault is thrown when one of the following errors occur:

- 2681 - A message is received for a particular conversation, after the conversation has ended
- 2682 - The conversation identification is invalid (not unique, out of range, etc.)
- 2683 - The conversation identification is not present in the input message of the operation that  
2684 ends the conversation
- 2685 - The client or the service attempts to send a message in a conversation, after the  
2686 conversation has ended

2687 This fault is named within the SCA namespace standard prefix "sca", which corresponds to URI  
2688 <http://docs.oasis-open.org/ns/opencsa/sca/200712>.

2689 The lifecycle of resources and the association between unique identifiers and conversations are  
2690 determined by the service's implementation type and may not be directly affected by the  
2691 "endConversation" annotation. For example, a **WS-BPEL** process **can** outlive most of the  
2692 conversations that it is involved in.

2693 Although conversational interfaces do not require that any identifying information be passed as  
2694 part of the body of messages, there is conceptually an identity associated with the conversation.

2695 Individual implementation types can have an API to access the ID associated with the  
2696 conversation, although no assumptions can be made about the structure of that identifier.  
2697 Implementation types can also have a means to set the conversation ID by either the client or the  
2698 service provider, although the operation may only be supported by some binding/policy  
2699 combinations.

2700 Implementation-type specifications are encouraged to define and provide conversational instance  
2701 lifecycle management for components that implement conversational interfaces. However,  
2702 implementations could also manage the conversational state manually.

2703

## 2704 **8.4 Long-running Request-Response Operations**

### 2705 **8.4.1 Background**

2706 A service offering one or more operations which map to a WSDL request-response pattern may be  
2707 implemented in a long-running, potentially interruptible, way. Consider a BPEL process with  
2708 receive and reply activities referencing the WSDL request-response operation. Between the two  
2709 activities, the business process logic may be a long-running sequence of steps, including activities  
2710 causing the process to be interrupted. Typical examples are steps where the process waits for  
2711 another message to arrive or a specified time interval to expire, or the process may perform  
2712 asynchronous interactions such as service invocations bound to asynchronous protocols or user  
2713 interactions. This is a common situation in business processes, and it causes the implementation  
2714 of the WSDL request-response operation to run for a very long time, e.g., several months (!). In  
2715 this case, it is not meaningful for any caller to remain in a synchronous wait for the response while  
2716 blocking system resources or holding database locks.

2717 Note that it is possible to model long-running interactions as a pair of two independent operations  
2718 as described in the section on bidirectional interfaces. However, it is a common practice (and in  
2719 fact much more convenient) to model a request-response operation and let the infrastructure deal  
2720 with the asynchronous message delivery and correlation aspects instead of putting this burden on  
2721 the application developer.

2722

### 2723 **8.4.2 Definition of "long-running"**

2724 A request-response operation is considered long-running if the implementation does not guarantee  
2725 the delivery of the response within any specified time interval. Clients invoking such request-  
2726 response operations are strongly discouraged from making assumptions about when the response  
2727 can be expected.

2728

### 2729 **8.4.3 The asyncInvocation Intent**

2730 This specification permits a long-running request-response operation or a complete interface  
2731 containing such operations to be marked using a policy intent with the name **asyncInvocation**. It  
2732 is also possible for a service to set the `asyncInvocation` intent when using an interface which is  
2733 not marked with the `asyncInvocation` intent. This can be useful when reusing an existing interface  
2734 definition that does not contain SCA information.

2735

### 2736 **8.4.4 Requirements on Bindings**

2737 In order to support a service operation which is marked with the `asyncInvocation` intent, it is  
2738 necessary for the binding (and its associated policies) to support separate handling of the request  
2739 message and the response message. Bindings which only support a synchronous style of message  
2740 handling, such as a conventional HTTP binding, cannot be used to support long-running  
2741 operations.

2742 The requirements on a binding to support the asyncInvocation intent are the same as those  
2743 required to support services with bidirectional interfaces - namely that the binding needs to be  
2744 able to treat the transmission of the request message separately from the transmission of the  
2745 response message, with an arbitrarily large time interval between the two transmissions.

2746 An example of a binding/policy combination that supports long-running request-response  
2747 operations is a Web service binding used in conjunction with the WS-Addressing  
2748 "wsam:NonAnonymousResponses" assertion.

2749

## 2750 8.4.5 Implementation Type Support

2751 SCA implementation types can provide special asynchronous client-side and asynchronous server-  
2752 side mappings to assist in the development of services and clients for long-running request-  
2753 response operations.

## 2754 8.5 SCA-Specific Aspects for WSDL Interfaces

2755 There are a number of aspects that SCA applies to interfaces in general, such as marking them  
2756 **conversational**. These aspects apply to the interfaces themselves, rather than their use in a  
2757 specific place within SCA. There is thus a need to provide appropriate ways of marking the  
2758 interface definitions themselves, which go beyond the basic facilities provided by the interface  
2759 definition language.

2760 For WSDL interfaces, there is an extension mechanism that permits additional information to be  
2761 included within the WSDL document. SCA takes advantage of this extension mechanism. In order  
2762 to use the SCA extension mechanism, the SCA namespace ([http://docs.oasis-](http://docs.oasis-open.org/ns/opencsa/sca/200712)  
2763 [open.org/ns/opencsa/sca/200712](http://docs.oasis-open.org/ns/opencsa/sca/200712)) needs to be declared within the WSDL document.

2764 First, SCA defines a global attribute in the SCA namespace which provides a mechanism to attach  
2765 policy intents - **@requires**. The definition of this attribute is as follows:

```
2766 <attribute name="requires" type="sca:listOfQNames"/>
```

2767

```
2768 <simpleType name="listOfQNames">
2769 <list itemType="QName"/>
2770 </simpleType>
```

2771 The @requires attribute can be applied to WSDL Port Type elements (WSDL 1.1). The attribute  
2772 contains one or more intent names, as defined by [the Policy Framework specification \[10\]](#). Any  
2773 service or reference that uses an interface marked with required intents MUST implicitly add those  
2774 intents to its own @requires list. [ASM80008]

2775 To specify that a WSDL interface is conversational, the following attribute setting is used on either  
2776 the WSDL Port Type or WSDL Interface:

```
2777 requires="conversational"
```

2778 SCA defines an **endsConversation** attribute that is used to mark specific operations within a  
2779 WSDL interface declaration as ending a conversation. This only has meaning for WSDL interfaces  
2780 which are also marked conversational. The endsConversation attribute is a global attribute in the  
2781 SCA namespace, with the following definition:

```
2782 <attribute name="endsConversation" type="boolean" default="false"/>
```

2783

2784 The following snippet is an example of a WSDL Port Type annotated with the **requires** attribute on  
2785 the portType and the **endsConversation** attribute on one of the operations:

```
2786 ...
2787 <portType name="LoanService" sca:requires="conversational">
2788 <operation name="apply">
2789 <input message="tns:ApplicationInput"/>
```

```

2790 <output message="tns:ApplicationOutput" />
2791 </operation>
2792 <operation name="cancel" sca:endsConversation="true">
2793 </operation>
2794 ...
2795 </portType>
2796 ...

```

2797 The following snippet is an example of a WSDL Port Type annotated with the **requires** attribute on  
2798 the portType and the **endsConversation** attribute on one of the operations:

```

2799 ...
2800 <portType name="LoanService" sca:requires="conversational">
2801 <operation name="apply">
2802 <input message="tns:ApplicationInput" />
2803 <output message="tns:ApplicationOutput" />
2804 </operation>
2805 <operation name="cancel" sca:endsConversation="true">
2806 </operation>
2807 ...
2808 </portType>
2809 ...

```

2810 SCA defines an attribute which is used to indicate that a given WSDL Port Type element (WSDL  
2811 1.1) has an associated callback interface. This is the @callback attribute, which applies to a WSDL  
2812 <portType/> element.

2813 The @callback attribute is defined as a global attribute in the SCA namespace, as follows:

```

2815 <attribute name="callback" type="QName" />

```

2816 The value of the @callback attribute is the QName of a Port Type. The port type declared by the  
2817 @callback attribute is the callback interface to use for the portType which is annotated by the  
2818 @callback attribute.

2820 Here is an example of a portType element with a callback attribute:

```

2822 <portType name="LoanService" sca:callback="foo:LoanServiceCallback">
2823 <operation name="apply">
2824 <input message="tns:ApplicationInput" />
2825 <output message="tns:ApplicationOutput" />
2826 </operation>
2827 ...
2828 </portType>

```

## 2830 8.6 WSDL Interface Type

2831 The WSDL interface type is used to declare interfaces for services and for references, where the interface  
2832 is defined in terms of a WSDL document. An interface is defined in terms of a WSDL 1.1 Port Type with  
2833 the arguments and return of the service operations described using XML schema.

2834 A WSDL interface is declared by an **interface.wsdl** element. The following shows the pseudo-schema  
2835 for the interface.wsdl element:

```

2837 <!-- WSDL Interface schema snippet -->
2838 <interface.wsdl interface="xs:anyURI" callbackInterface="xs:anyURI"?>

```

2839 The interface.wsdl element has the following **attributes**:

- 2840 • **interface (1..1)** - the URI of a WSDL Port Type

2841 The interface.wSDL @interface attribute MUST reference a portType of a WSDL 1.1  
2842 document. [ASM80001]

- 2843 • **callbackInterface(0..1)** - an optional callback interface, which is the URI of a WSDL Port  
2844 Type

2845 The interface.wSDL @callbackInterface attribute, if present, MUST reference a portType of a  
2846 WSDL 1.1 document. [ASM80016]

2847

2848 The form of the URI for WSDL port types follows the syntax described in the WSDL 1.1 Element  
2849 Identifiers specification [WSDL11\_Identifiers]

### 2850 **8.6.1 Example of interface.wSDL**

```
2851 <interface.wSDL interface="http://www.stockquote.org/StockQuoteService#
2852 wsdl.porttype(StockQuote)"
2853 callbackInterface="http://www.stockquote.org/StockQuoteService#
2854 wsdl.porttype(StockQuoteCallback)"/>
2855
```

2856 This declares an interface in terms of the WSDL port type "StockQuote" with a callback interface defined  
2857 by the "StockQuoteCallback" port type.

2858

## 9 Binding

2859

2860 Bindings are used by services and references. References use bindings to describe the access  
2861 mechanism used to call a service (which can be a service provided by another SCA composite).  
2862 Services use bindings to describe the access mechanism that clients (which can be a client from  
2863 another SCA composite) have to use to call the service.

2864 SCA supports the use of multiple different types of bindings. Examples include **SCA service, Web**  
2865 **service, stateless session EJB, data base stored procedure, EIS service**. An SCA runtime  
2866 MUST provide support for SCA service and Web service binding types. SCA provides an  
2867 extensibility mechanism by which an SCA runtime can add support for additional binding types.  
2868 For details on how additional binding types are defined, see the section on the Extension Model.

2869 A binding is defined by a **binding element** which is a child element of a service or of a reference  
2870 element in a composite. The following snippet shows the composite schema with the schema for  
2871 the binding element.

2872

```
2873 <?xml version="1.0" encoding="ASCII"?>
2874 <!-- Bindings schema snippet -->
2875 <composite ... >
2876 ...
2877 <service ... >*
2878 <interface ... />?
2879 <binding uri="xs:anyURI"? name="xs:NCName"?
2880 requires="list of xs:QName"?
2881 policySets="list of xs:QName"?>*
2882 <operation name="xs:NCName" requires="list of xs:QName"?
2883 policySets="list of xs:QName"?/>*
2884 <wireFormat/>?
2885 <operationSelector/>?
2886 </binding>
2887 <callback?
2888 <binding uri="xs:anyURI"? name="xs:NCName"?
2889 requires="list of xs:QName"?
2890 policySets="list of xs:QName"?>+
2891 <operation name="xs:NCName" requires="list of xs:QName"?
2892 policySets="list of xs:QName"?/>*
2893 <wireFormat/>?
2894 <operationSelector/>?
2895 </binding>
2896 </callback>
2897 </service>
2898 ...
2899 <reference ... >*
2900 <interface ... />?
2901 <binding uri="xs:anyURI"? name="xs:NCName"?
2902 requires="list of xs:QName"?
2903 policySets="list of xs:QName"?>*
2904 <operation name="xs:NCName" requires="list of xs:QName"?
2905 policySets="list of xs:QName"?/>*
2906 <wireFormat/>?
2907 <operationSelector/>?
2908 </binding>
2909 <callback?
2910 <binding uri="xs:anyURI"? name="xs:NCName"?
2911 requires="list of xs:QName"?
```

```

2912 policySets="list of xs:QName"?>+
2913 <operation name="xs:NCName" requires="list of xs:QName"?
2914 policySets="list of xs:QName"?/>*
2915 <wireFormat/>?
2916 <operationSelector/>?
2917 </binding>
2918 </callback>
2919 </reference>
2920 ...
2921 </composite>
2922

```

2923 The element name of the binding element is architected; it is in itself a qualified name. The first  
 2924 qualifier is always named "binding", and the second qualifier names the respective binding-type  
 2925 (e.g. binding.composite, binding.ws, binding.ejb, binding.eis).

2926

2927 A binding element has the following attributes:

- 2928 • **uri (0..1)** - has the following semantic.
  - 2929 ○ The uri attribute can be omitted.
  - 2930 ○ For a binding of a **reference** the URI attribute defines the target URI of the
  - 2931 reference. This MUST be either the componentName/serviceName for a wire to an
  - 2932 endpoint within the SCA domain, or the accessible address of some service
  - 2933 endpoint either inside or outside the SCA domain (where the addressing scheme is
  - 2934 defined by the type of the binding). [\[ASM90001\]](#)
  - 2935 ○ The circumstances under which the uri attribute can be used are defined in
  - 2936 section ["5.3.1 Specifying the Target Service\(s\) for a Reference."](#)
  - 2937 ○ For a binding of a **service** the URI attribute defines the URI relative to the
  - 2938 component, which contributes the service to the SCA domain. The default value for
  - 2939 the URI is the value of the name attribute of the binding.
- 2940 • **name (0..1)** – a name for the binding instance (an NCName). The name attribute allows
- 2941 distinction between multiple binding elements on a single service or reference. The
- 2942 default value of the name attribute is the service or reference name. When a service or
- 2943 reference has multiple bindings, only one binding can have the default name value; all
- 2944 others must have a name value specified that is unique within the service or reference.
- 2945 [\[ASM90002\]](#) The name also permits the binding instance to be referenced from elsewhere
- 2946 – particularly useful for some types of binding, which can be declared in a definitions
- 2947 document as a template and referenced from other binding instances, simplifying the
- 2948 definition of more complex binding instances (see [the JMS Binding specification \[11\]](#) for
- 2949 examples of this referencing).
- 2950 • **requires (0..1)** - a list of policy intents. See the [Policy Framework specification \[10\]](#) for a
- 2951 description of this attribute.
- 2952 • **policySets (0..1)** – a list of policy sets. See the [Policy Framework specification \[10\]](#) for a
- 2953 description of this attribute.

2954 A binding element has the following child elements:

- 2955 • **operation: Operation (0..n)** - Zero or more operation elements. These elements are
- 2956 used to describe characteristics of individual operations within the interface. For a detailed
- 2957 description of the operation element, see [the Policy Framework specification \[SCA Policy\]](#).
- 2958 • **wireFormat (0..1)** - a wireFormat to apply to the data flowing using the binding. See [the](#)
- 2959 [wireFormat section](#) for details.
- 2960 • **operationSelector(0..1)** - an operationSelector element that is used to match a
- 2961 particular message to a particular operation in the interface. See [the operationSelector](#)
- 2962 [section](#) for details

2963 When multiple bindings exist for an service, it means that the service is available by any of the  
2964 specified bindings. The technique that the SCA runtime uses to choose among available bindings  
2965 is left to the implementation and it may include additional (nonstandard) configuration. Whatever  
2966 technique is used needs to be documented by the runtime.

2967 Services and References can always have their bindings overridden at the SCA domain level,  
2968 unless restricted by Intents applied to them.

2969 If a reference has any bindings they MUST be resolved which means that each binding MUST  
2970 include a value for the @URI attribute or MUST otherwise specify an endpoint. The reference  
2971 MUST NOT be wired using other SCA mechanisms. [ASM90003] To specify constraints on the kinds  
2972 of bindings that are acceptable for use with a reference, the user specifies either policy intents or  
2973 policy sets.

2974 Users can also specifically wire, not just to a component service, but to a specific binding offered  
2975 by that target service. To do so, a wire target MAY be specified with a syntax of  
2976 "componentName/serviceName/bindingName". [ASM90004]  
2977

2978

2979 The following sections describe the SCA and Web service binding type in detail.

2980

## 2981 9.1 Messages containing Data not defined in the Service Interface

2982 It is possible for a message to include information that is not defined in the interface used to  
2983 define the service, for instance information may be contained in SOAP headers or as MIME  
2984 attachments.

2985 Implementation types can make this information available to component implementations in their  
2986 execution context. The specifications for these implementation types describe how this  
2987 information is accessed and in what form it is presented.

2988

## 2989 9.2 WireFormat

2990 A wireFormat is the form that a data structure takes when it is transmitted using some  
2991 communication binding. Another way to describe this is "the form that the data takes on the wire".  
2992 A wireFormat can be specific to a given communication method, or it may be general, applying to  
2993 many different communication methods. An example of a general wireFormat is XML text format.

2994 Where a particular SCA binding can accommodate transmitting data in more than one format, the  
2995 configuration of the binding MAY include a definition of the wireFormat to use. This is done using  
2996 an optional <sca:wireFormat/> subelement of the <binding/> element.

2997 Where a binding supports more than one wireFormat, the binding defines one of the wireFormats  
2998 to be the default wireFormat which applies if no <wireFormat/> subelement is present.

2999 The base sca:wireFormat element is abstract and it has no attributes and no child elements. For a  
3000 particular wireFormat, an extension subtype is defined, using substitution groups, for example:

- 3001 • <sca:wireFormat.xml/>
- 3002 • A wireFormat that transmits the data as an XML text datastructure
- 3003 • <sca:wireFormat.jms/>
- 3004 • The "default JMS wireFormat" as described in the JMS Binding specification

3005

3006 Specific wireFormats can have elements that include either attributes or subelements or both.

3007 For details about specific wireFormats, see the related SCA Binding specifications.

3008

## 3009 9.3 OperationSelector

3010 An operationSelector is necessary for some types of transport binding where messages are  
3011 transmitted across the transport without any explicit relationship between the message and the  
3012 interface operation to which it relates. SOAP is an example of a protocol where the messages do  
3013 contain explicit information that relates each message to the operation it targets. However, other  
3014 transport bindings have messages where this relationship is not expressed in the message or in  
3015 any related headers (pure JMS messages, for example). In cases where the messages arrive at a  
3016 service without any explicit information that maps them to specific operations, it is necessary for  
3017 the metadata attached to the service binding to contain the required mapping information. The  
3018 information is held in an operationSelector element which is a child element of the binding  
3019 element.

3020 The base `sca:operationSelector` element is abstract and it has no attributes and no child elements.  
3021 For a particular operationSelector, an extension subtype is defined, using substitution groups, for  
3022 example:

3023 

- `<sca:operationSelector.XPath/>`

3024 

- An operation selector that uses XPath to filter out specific messages and target them to  
3025 particular named operations.

3026

3027 Specific operationSelectors can have elements that include either attributes or subelements or  
3028 both.

3029 For details about specific operationSelectors, see the related SCA Binding specifications.

3030

## 3031 9.4 Form of the URI of a Deployed Binding

3032 SCA Bindings specifications can choose to use the **structural URI** defined in the section  
3033 "[Structural URI of Components](#)" above to derive a binding specific URI according to some Binding-  
3034 related scheme. The relevant binding specification describes this.

3035 Alternatively, `<binding/>` elements have an optional `@URI` attribute, which is termed a  
3036 bindingURI.

3037 If the bindingURI is specified on a given `<binding/>` element, the binding can optionally use it to  
3038 derive an endpoint URI relevant to the binding. The derivation is binding specific and is described  
3039 by the relevant binding specification.

3040 For `binding.sca`, which is described in the SCA Assembly specification, this is as follows:

3041 

- If the binding uri attribute is specified on a reference, it identifies the target service in the  
3042 SCA domain by specifying the service's structural URI.

3043 

- If the binding uri attribute is specified on a service, it is ignored.

3044

### 3045 9.4.1 Non-hierarchical URIs

3046 Bindings that use non-hierarchical URI schemes (such as `jms:` or `mailto:`) may optionally make  
3047 use of the "uri" attribute, which is the complete representation of the URI for that service  
3048 binding. Where the binding does not use the "uri" attribute, the binding needs to offer a different  
3049 mechanism for specifying the service address.

### 3050 9.4.2 Determining the URI scheme of a deployed binding

3051 One of the things that needs to be determined when building the effective URI of a deployed  
3052 binding (i.e. endpoint) is the URI scheme. The process of determining the endpoint URI scheme is  
3053 binding type specific.

3054 If the binding type supports a single protocol then there is only one URI scheme associated with it.  
3055 In this case, that URI scheme is used.

3056 If the binding type supports multiple protocols, the binding type implementation determines the  
3057 URI scheme by introspecting the binding configuration, which may include the policy sets  
3058 associated with the binding.

3059 A good example of a binding type that supports multiple protocols is binding.ws, which can be  
3060 configured by referencing either an "abstract" WSDL element (i.e. portType or interface) or a  
3061 "concrete" WSDL element (i.e. binding, port or endpoint). When the binding references a PortType  
3062 or Interface, the protocol and therefore the URI scheme is derived from the intents/policy sets  
3063 attached to the binding. When the binding references a "concrete" WSDL element, there are two  
3064 cases:

3065 1) The referenced WSDL binding element uniquely identifies a URI scheme. This is the most  
3066 common case. In this case, the URI scheme is given by the protocol/transport specified in the  
3067 WSDL binding element.

3068 2) The referenced WSDL binding element doesn't uniquely identify a URI scheme. For example,  
3069 when HTTP is specified in the @transport attribute of the SOAP binding element, both "http"  
3070 and "https" could be used as valid URI schemes. In this case, the URI scheme is determined  
3071 by looking at the policy sets attached to the binding.

3072 It's worth noting that an intent supported by a binding type may completely change the behavior  
3073 of the binding. For example, when the intent "confidentiality/transport" is required by an HTTP  
3074 binding, SSL is turned on. This basically changes the URI scheme of the binding from "http" to  
3075 "https".

3076

## 3077 9.5 SCA Binding

3078 The SCA binding element is defined by the following schema.

3079

3080 `<binding.sca />`

3081

3082 The SCA binding can be used for service interactions between references and services contained  
3083 within the SCA domain. The way in which this binding type is implemented is not defined by the  
3084 SCA specification and it can be implemented in different ways by different SCA runtimes. The only  
3085 requirement is that the required qualities of service must be implemented for the SCA binding  
3086 type. The SCA binding type is **not** intended to be an interoperable binding type. For  
3087 interoperability, an interoperable binding type such as the Web service binding should be used.

3088 A service definition with no binding element specified uses the SCA binding.

3089 `<binding.sca/>` would only have to be specified in override cases, or when you specify a  
3090 set of bindings on a service definition and the SCA binding should be one of them.

3091 If a reference does not have a binding, then the binding used can be any of the bindings  
3092 specified by the service provider, as long as the intents required by the reference and  
3093 the service are all respected.

3094 If the interface of the service or reference is local, then the local variant of the SCA  
3095 binding will be used. If the interface of the service or reference is remotable, then either  
3096 the local or remote variant of the SCA binding will be used depending on whether source  
3097 and target are co-located or not.

3098 If a reference specifies an URI via its uri attribute, then this provides the default wire to a service  
3099 provided by another domain level component. The value of the URI has to be as follows:

- 3100 • `<domain-component-name>/<service-name>`

3101

## 3102 9.5.1 Example SCA Binding

3103 The following snippet shows the MyValueComposite.composite file for the MyValueComposite  
3104 containing the service element for the MyValueService and a reference element for the  
3105 StockQuoteService. Both the service and the reference use an SCA binding. The target for the  
3106 reference is left undefined in this binding and would have to be supplied by the composite in which  
3107 this composite is used.

3108

```
3109 <?xml version="1.0" encoding="ASCII"?>
3110 <!-- Binding SCA example -->
3111 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3112 targetNamespace="http://foo.com"
3113 name="MyValueComposite" >
3114
3115 <service name="MyValueService" promote="MyValueComponent">
3116 <interface.java interface="services.myvalue.MyValueService"/>
3117 <binding.sca/>
3118 ...
3119 </service>
3120
3121 ...
3122
3123 <reference name="StockQuoteService"
3124 promote="MyValueComponent/StockQuoteReference">
3125 <interface.java interface="services.stockquote.StockQuoteService"/>
3126 <binding.sca/>
3127 </reference>
3128
3129 </composite>
3130
```

## 3131 9.6 Web Service Binding

3132 SCA defines a Web services binding. This is described in [a separate specification document \[9\]](#).

3133

## 3134 9.7 JMS Binding

3135 SCA defines a JMS binding. This is described in [a separate specification document \[11\]](#).

---

## 10 SCA Definitions

3136  
3137  
3138  
3139  
3140  
3141  
3142  
3143  
3144  
3145  
3146  
3147  
3148  
3149  
3150  
3151  
3152  
3153  
3154  
3155  
3156  
3157  
3158  
3159  
3160  
3161  
3162  
3163  
3164  
3165  
3166  
3167  
3168  
3169  
3170

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

All of these artifacts within an SCA Domain are defined in SCA contributions in files called META-INF/definitions.xml (relative to the contribution base URI). Although the definitions are specified within a single SCA contribution, the definitions are visible throughout the domain. Because of this, all of the QNames for the definitions contained in definitions.xml files MUST be unique within the domain. [ASM10001] The definitions.xml file contains a definitions element that conforms to the following pseudo-schema snippet:

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

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

</definitions>
```

The definitions element has the following attribute:

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

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

## 3171 11 Extension Model

3172  
3173 The assembly model can be extended with support for new interface types, implementation types  
3174 and binding types. The extension model is based on XML schema substitution groups. There are  
3175 three XML Schema substitution group heads defined in the SCA namespace: **interface**,  
3176 **implementation** and **binding**, for interface types, implementation types and binding types,  
3177 respectively.

3178 The SCA Client and Implementation specifications and the SCA Bindings specifications (see [1],  
3179 [9], [11]) use these XML Schema substitution groups to define some basic types of interfaces,  
3180 implementations and bindings, but other types can be defined as required, where support for  
3181 these extra ones is available from the runtime. The interface type elements, implementation type  
3182 elements, and binding type elements defined by the SCA specifications are all part of the SCA  
3183 namespace ("http://docs.oasis-open.org/ns/opencsa/sca/200712"), as indicated in their  
3184 respective schemas. New interface types, implementation types and binding types that are defined  
3185 using this extensibility model, which are not part of these SCA specifications are defined in  
3186 namespaces other than the SCA namespace.

3187 The "." notation is used in naming elements defined by the SCA specifications ( e.g.  
3188 <implementation.java ... />, <interface.wsdl ... />, <binding.ws ... />), not as a parallel  
3189 extensibility approach but as a naming convention that improves usability of the SCA assembly  
3190 language.

3191  
3192 **Note:** How to contribute SCA model extensions and their runtime function to an SCA runtime will  
3193 be defined by a future version of the specification.

3194

### 3195 11.1 Defining an Interface Type

3196 The following snippet shows the base definition for the **interface** element and **Interface** type  
3197 contained in **sca-core.xsd**; see appendix for complete schema.

```
3198
3199 <?xml version="1.0" encoding="UTF-8"?>
3200 <!-- (c) Copyright SCA Collaboration 2006 -->
3201 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3202 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3203 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3204 elementFormDefault="qualified">
```

3205

3206 ...

3207

```
3208 <element name="interface" type="sca:Interface" abstract="true"/>
3209 <complexType name="Interface"/>
3210 <complexType name="Interface" abstract="true">
3211 <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3212 <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3213 </complexType>
```

3214

3215  
3216  
3217  
3218  
3219  
3220  
3221  
3222  
3223  
3224  
3225  
3226  
3227  
3228  
3229  
3230  
3231  
3232  
3233  
3234  
3235  
3236  
3237  
3238  
3239  
3240  
3241  
3242  
3243  
3244  
3245  
3246  
3247  
3248  
3249  
3250  
3251  
3252  
3253  
3254  
3255  
3256  
3257  
3258

```
...
</schema>
```

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

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

 <element name="interface.java" type="sca:JavaInterface"
 substitutionGroup="sca:interface"/>
 <complexType name="JavaInterface">
 <complexContent>
 <extension base="sca:Interface">
 <attribute name="interface" type="NCName"
 use="required"/>
 </extension>
 </complexContent>
 </complexType>
</schema>
```

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

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

 <element name="my-interface-extension"
 type="tns:my-interface-extension-type"
 substitutionGroup="sca:interface"/>
 <complexType name="my-interface-extension-type">
 <complexContent>
 <extension base="sca:Interface">
 ...
 </extension>
 </complexContent>
 </complexType>
```

3259 </schema>  
3260

## 3261 11.2 Defining an Implementation Type

3262 The following snippet shows the base definition for the *implementation* element and  
3263 *Implementation* type contained in *sca-core.xsd*; see appendix for complete schema.

```
3264
3265 <?xml version="1.0" encoding="UTF-8"?>
3266 <!-- (c) Copyright SCA Collaboration 2006 -->
3267 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3268 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3269 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3270 elementFormDefault="qualified">
3271
3272 ...
3273
3274 <element name="implementation" type="sca:Implementation"
3275 abstract="true"/>
3276 <complexType name="Implementation"/>
3277
3278 ...
3279
3280 </schema>
```

3281  
3282 In the following snippet we show how the base definition is extended to support Java  
3283 implementation. The snippet shows the definition of the *implementation.java* element and the  
3284 *JavaImplementation* type contained in *sca-implementation-java.xsd*.

```
3285
3286 <?xml version="1.0" encoding="UTF-8"?>
3287 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3288 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3289 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712">
3290
3291 <element name="implementation.java" type="sca:JavaImplementation"
3292 substitutionGroup="sca:implementation"/>
3293 <complexType name="JavaImplementation">
3294 <complexContent>
3295 <extension base="sca:Implementation">
3296 <attribute name="class" type="NCName"
3297 use="required"/>
3298 </extension>
3299 </complexContent>
3300 </complexType>
3301 </schema>
```

3302 In the following snippet is an example of how the base definition can be extended by other  
3303 specifications to support a new implementation type not defined in the SCA specifications. The  
3304 snippet shows the definition of the **my-impl-extension** element and the **my-impl-extension-**  
3305 **type** type.

```
3306 <?xml version="1.0" encoding="UTF-8"?>
3307 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3308 targetNamespace="http://www.example.org/myextension"
3309 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3310 xmlns:tns="http://www.example.org/myextension">
3311
3312 <element name="my-impl-extension" type="tns:my-impl-extension-type"
3313 substitutionGroup="sca:implementation"/>
3314 <complexType name="my-impl-extension-type">
3315 <complexContent>
3316 <extension base="sca:Implementation">
3317 ...
3318 </extension>
3319 </complexContent>
3320 </complexType>
3321 </schema>
3322
```

3323 In addition to the definition for the new implementation instance element, there needs to be an  
3324 associated implementationType element which provides metadata about the new implementation  
3325 type. The pseudo schema for the implementationType element is shown in the following snippet:

```
3326 <implementationType type="xs:QName"
3327 alwaysProvides="list of intent xs:QName"
3328 mayProvide="list of intent xs:QName"/>
3329
```

3330 The implementation type has the following attributes:

- 3331 • **type (1..1)** – the type of the implementation to which this implementationType element  
3332 applies. This is intended to be the QName of the implementation element for the  
3333 implementation type, such as "sca:implementation.java"
  - 3334 • **alwaysProvides (0..1)** – a set of intents which the implementation type always  
3335 provides. See [the Policy Framework specification \[10\]](#) for details.
  - 3336 • **mayProvide (0..1)** – a set of intents which the implementation type may provide. See  
3337 [the Policy Framework specification \[10\]](#) for details.
- 3338

## 3339 11.3 Defining a Binding Type

3340 The following snippet shows the base definition for the **binding** element and **Binding** type  
3341 contained in **sca-core.xsd**; see appendix for complete schema.

```
3342
3343 <?xml version="1.0" encoding="UTF-8"?>
3344 <!-- binding type schema snippet -->
3345 <!-- (c) Copyright SCA Collaboration 2006, 2007 -->
```

```

3346 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3347 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3348 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3349 elementFormDefault="qualified">
3350
3351 ...
3352
3353 <element name="binding" type="sca:Binding" abstract="true"/>
3354 <complexType name="Binding">
3355 <attribute name="uri" type="anyURI" use="optional"/>
3356 <attribute name="name" type="NCName" use="optional"/>
3357 <attribute name="requires" type="sca:listOfQNames"
3358 use="optional"/>
3359 <attribute name="policySets" type="sca:listOfQNames"
3360 use="optional"/>
3361 </complexType>
3362
3363 ...
3364
3365 </schema>

```

3366 In the following snippet is an example of how the base definition is extended to support Web  
3367 service binding. The snippet shows the definition of the **binding.ws** element and the  
3368 **WebServiceBinding** type contained in **sca-binding-webservice.xsd**.

```

3369
3370 <?xml version="1.0" encoding="UTF-8"?>
3371 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3372 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3373 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712">
3374
3375 <element name="binding.ws" type="sca:WebServiceBinding"
3376 substitutionGroup="sca:binding"/>
3377 <complexType name="WebServiceBinding">
3378 <complexContent>
3379 <extension base="sca:Binding">
3380 <attribute name="port" type="anyURI" use="required"/>
3381 </extension>
3382 </complexContent>
3383 </complexType>
3384 </schema>

```

3385 In the following snippet is an example of how the base definition can be extended by other  
3386 specifications to support a new binding not defined in the SCA specifications. The snippet shows  
3387 the definition of the **my-binding-extension** element and the **my-binding-extension-type** type.

```

3388 <?xml version="1.0" encoding="UTF-8"?>
3389 <schema xmlns="http://www.w3.org/2001/XMLSchema"

```

```

3390 targetNamespace="http://www.example.org/myextension"
3391 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3392 xmlns:tns="http://www.example.org/myextension">
3393
3394 <element name="my-binding-extension"
3395 type="tns:my-binding-extension-type"
3396 substitutionGroup="sca:binding"/>
3397 <complexType name="my-binding-extension-type">
3398 <complexContent>
3399 <extension base="sca:Binding">
3400 ...
3401 </extension>
3402 </complexContent>
3403 </complexType>
3404 </schema>
3405

```

3406 In addition to the definition for the new binding instance element, there needs to be an associated  
3407 bindingType element which provides metadata about the new binding type. The pseudo schema  
3408 for the bindingType element is shown in the following snippet:

```

3409 <bindingType type="xs:QName"
3410 alwaysProvides="list of intent QNames"?
3411 mayProvide = "list of intent QNames"?/>
3412

```

3413 The binding type has the following attributes:

- 3414 • **type (1..1)** – the type of the binding to which this bindingType element applies. This is  
3415 intended to be the QName of the binding element for the binding type, such as  
3416 "sca:binding.ws"
- 3417 • **alwaysProvides (0..1)** – a set of intents which the binding type always provides. See  
3418 [the Policy Framework specification \[10\]](#) for details.
- 3419 • **mayProvide (0..1)** – a set of intents which the binding type may provide. See [the](#)  
3420 [Policy Framework specification \[10\]](#) for details.

## 3421 11.4 Defining an Import Type

3422 The following snippet shows the base definition for the *import* element and *Import* type contained in *sca-*  
3423 *core.xsd*; see appendix for complete schema.

```

3424
3425 <?xml version="1.0" encoding="UTF-8"?>
3426 <!-- Copyright(C) OASIS(R) 2005,2008. All Rights Reserved. OASIS trademark,
3427 IPR and other policies apply. -->
3428 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3429 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3430 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3431 elementFormDefault="qualified">
3432
3433 ...
3434
3435 <!-- Import -->
3436 <element name="importBase" type="sca:Import" abstract="true" />

```

```

3437 <complexType name="Import" abstract="true">
3438 <complexContent>
3439 <extension base="sca:CommonExtensionBase">
3440 <sequence>
3441 <any namespace="##other" processContents="lax" minOccurs="0"
3442 maxOccurs="unbounded" />
3443 </sequence>
3444 </extension>
3445 </complexContent>
3446 </complexType>
3447
3448 <element name="import" type="sca:ImportType"
3449 substitutionGroup="sca:importBase" />
3450 <complexType name="ImportType">
3451 <complexContent>
3452 <extension base="sca:Import">
3453 <attribute name="namespace" type="string" use="required" />
3454 <attribute name="location" type="anyURI" use="required" />
3455 </extension>
3456 </complexContent>
3457 </complexType>
3458
3459 ...
3460
3461 </schema>

```

3462

3463 In the following snippet we show how the base import definition is extended to support Java imports. In  
3464 the import element, the namespace is expected to be an XML namespace, an import.java element uses a  
3465 Java package name instead. The snippet shows the definition of the *import.java* element and the  
3466 *JavaImportType* type contained in *sca-import-java.xsd*.

```

3467
3468 <?xml version="1.0" encoding="UTF-8"?>
3469 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3470 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3471 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712">
3472
3473 <element name="import.java" type="sca:JavaImportType"
3474 substitutionGroup="sca:importBase" />
3475 <complexType name="JavaImportType">
3476 <complexContent>
3477 <extension base="sca:Import">
3478 <attribute name="package" type="xs:String" use="required" />
3479 <attribute name="location" type="xs:AnyURI" use="optional" />
3480 </extension>
3481 </complexContent>
3482 </complexType>
3483 </schema>

```

3484

3485 In the following snippet we show an example of how the base definition can be extended by other  
3486 specifications to support a new interface not defined in the SCA specifications. The snippet shows the  
3487 definition of the *my-import-extension* element and the *my-import-extension-type* type.

```

3488
3489 <?xml version="1.0" encoding="UTF-8"?>
3490 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3491 targetNamespace="http://www.example.org/myextension"

```

```

3492 xmlns:sca=" http://docs.oasis-open.org/ns/opencsa/sca/200712"
3493 xmlns:tns="http://www.example.org/myextension">
3494
3495 <element name="my-import-extension"
3496 type="tns:my-import-extension-type"
3497 substitutionGroup="sca:importBase"/>
3498 <complexType name="my-import-extension-type">
3499 <complexContent>
3500 <extension base="sca:Import">
3501 ...
3502 </extension>
3503 </complexContent>
3504 </complexType>
3505 </schema>

```

3507 For a complete example using this extension point, see the definition of *import.java* in the SCA Java  
3508 Common Annotations and APIs Specification [SCA-Java].

## 3509 11.5 Defining an Export Type

3510 The following snippet shows the base definition for the *export* element and *ExportType* type contained in  
3511 *sca-core.xsd*; see appendix for complete schema.

```

3512
3513 <?xml version="1.0" encoding="UTF-8"?>
3514 <!-- Copyright(C) OASIS(R) 2005,2008. All Rights Reserved. OASIS trademark,
3515 IPR and other policies apply. -->
3516 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3517 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3518 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3519 elementFormDefault="qualified">
3520
3521 ...
3522 <!-- Export -->
3523 <element name="exportBase" type="sca:Export" abstract="true" />
3524 <complexType name="Export" abstract="true">
3525 <complexContent>
3526 <extension base="sca:CommonExtensionBase">
3527 <sequence>
3528 <any namespace="##other" processContents="lax" minOccurs="0"
3529 maxOccurs="unbounded"/>
3530 </sequence>
3531 </extension>
3532 </complexContent>
3533 </complexType>
3534
3535 <element name="export" type="sca:ExportType"
3536 substitutionGroup="sca:exportBase"/>
3537 <complexType name="ExportType">
3538 <complexContent>
3539 <extension base="sca:Export">
3540 <attribute name="namespace" type="string" use="required"/>
3541 </extension>
3542 </complexContent>
3543 </complexType>
3544 ...
3545 </schema>

```

3546

3547 The following snippet shows how the base definition is extended to support Java exports. In a base  
3548 *export* element, the *@namespace* attribute specifies XML namespace being exported. An *export.java*  
3549 element uses a *@package* attribute to specify the Java package to be exported. The snippet shows the  
3550 definition of the ***export.java*** element and the ***JavaExport*** type contained in ***sca-export-java.xsd***.

3551

```
3552 <?xml version="1.0" encoding="UTF-8"?>
3553 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3554 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3555 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712">
3556
3557 <element name="export.java" type="sca:JavaExportType"
3558 substitutionGroup="sca:exportBase"/>
3559 <complexType name="JavaExportType">
3560 <complexContent>
3561 <extension base="sca:Export">
3562 <attribute name="package" type="xs:String" use="required"/>
3563 </extension>
3564 </complexContent>
3565 </complexType>
3566 </schema>
```

3567

3568 In the following snippet we show an example of how the base definition can be extended by other  
3569 specifications to support a new interface not defined in the SCA specifications. The snippet shows the  
3570 definition of the ***my-export-extension*** element and the ***my-export-extension-type*** type.

3571

```
3572 <?xml version="1.0" encoding="UTF-8"?>
3573 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3574 targetNamespace="http://www.example.org/myextension"
3575 xmlns:sca="http:// docs.oasis-open.org/ns/opencsa/sca/200712"
3576 xmlns:tns="http://www.example.org/myextension">
3577
3578 <element name="my-export-extension"
3579 type="tns:my-export-extension-type"
3580 substitutionGroup="sca:exportBase"/>
3581 <complexType name="my-export-extension-type">
3582 <complexContent>
3583 <extension base="sca:Export">
3584 ...
3585 </extension>
3586 </complexContent>
3587 </complexType>
3588 </schema>
```

3589

3590 For a complete example using this extension point, see the definition of ***export.java*** in the SCA Java  
3591 Common Annotations and APIs Specification [SCA-Java].

3592

---

## 3593 12 Packaging and Deployment

### 3594 12.1 Domains

3595 An **SCA Domain** represents a complete runtime configuration, potentially distributed over a series  
3596 of interconnected runtime nodes.

3597 A single SCA domain defines the boundary of visibility for all SCA mechanisms. For example, SCA  
3598 wires can only be used to connect components within a single SCA domain. Connections to  
3599 services outside the domain must use binding specific mechanisms for addressing services (such  
3600 as WSDL endpoint URIs). Also, SCA mechanisms such as intents and policySets can only be used  
3601 in the context of a single domain. In general, external clients of a service that is developed and  
3602 deployed using SCA should not be able to tell that SCA was used to implement the service – it is  
3603 an implementation detail.

3604 The size and configuration of an SCA Domain is not constrained by the SCA Assembly specification  
3605 and is expected to be highly variable. An SCA Domain typically represents an area of business  
3606 functionality controlled by a single organization. For example, an SCA Domain may be the whole  
3607 of a business, or it may be a department within a business.

3608 As an example, for the accounts department in a business, the SCA Domain might cover all  
3609 finance-related functions, and it might contain a series of composites dealing with specific areas of  
3610 accounting, with one for Customer accounts and another dealing with Accounts Payable.

3611 An SCA domain has the following:

- 3612 • A virtual domain-level composite whose components are deployed and running
- 3613 • A set of *installed contributions* that contain implementations, interfaces and other artifacts  
3614 necessary to execute components
- 3615 • A set of logical services for manipulating the set of contributions and the virtual domain-  
3616 level composite.

3617 The information associated with an SCA domain can be stored in many ways, including but not  
3618 limited to a specific filesystem structure or a repository.

### 3619 12.2 Contributions

3620 An SCA domain might require a large number of different artifacts in order to work. These  
3621 artifacts include artifacts defined by SCA and other artifacts such as object code files and interface  
3622 definition files. The SCA-defined artifact types are all XML documents. The root elements of the  
3623 different SCA definition documents are: composite, componentType, constrainingType and  
3624 definitions. XML artifacts that are not defined by SCA but which may be needed by an SCA  
3625 domain include XML Schema documents, WSDL documents, and BPEL documents. SCA  
3626 constructs, like other XML-defined constructs, use XML qualified names for their identity (i.e.  
3627 namespace + local name).

3628 Non-XML artifacts are also required within an SCA domain. The most obvious examples of such  
3629 non-XML artifacts are Java, C++ and other programming language files necessary for component  
3630 implementations. Since SCA is extensible, other XML and non-XML artifacts may also be required.

3631 SCA defines an interoperable packaging format for contributions (ZIP), as specified below. This  
3632 format is not the only packaging format that an SCA runtime can use. SCA allows many different  
3633 packaging formats, but requires that the ZIP format be supported. When using the ZIP format for  
3634 deploying a contribution, this specification does not specify whether that format is retained after  
3635 deployment. For example, a Java EE based SCA runtime may convert the ZIP package to an EAR  
3636 package. SCA expects certain characteristics of any packaging:

- 3637 • For any contribution packaging it MUST be possible to present the artifacts of the  
3638 packaging to SCA as a hierarchy of resources based off of a single root [\[ASM12001\]](#)

- 3639           • Within any contribution packaging A directory resource SHOULD exist at the root of the  
3640 hierarchy named META-INF [ASM12002]
- 3641           • Within any contribution packaging a document SHOULD exist directly under the META-INF  
3642 directory named sca-contribution.xml which lists the SCA Composites within the  
3643 contribution that are runnable. [ASM12003]  
3644
- 3645           The same document also optionally lists namespaces of constructs that are defined within  
3646 the contribution and which may be used by other contributions  
3647           Optionally, in the sca-contribution.xml file, additional elements MAY exist that list the  
3648 namespaces of constructs that are needed by the contribution and which are be found  
3649 elsewhere, for example in other contributions. [ASM12004] These optional elements may  
3650 not be physically present in the packaging, but may be generated based on the definitions  
3651 and references that are present, or they may not exist at all if there are no unresolved  
3652 references.  
3653
- 3654           See the section "SCA Contribution Metadata Document" for details of the format of this  
3655 file.

3656           To illustrate that a variety of packaging formats can be used with SCA, the following are examples  
3657 of formats that might be used to package SCA artifacts and metadata (as well as other artifacts)  
3658 as a contribution:

- 3659           • A filesystem directory  
3660           • An OSGi bundle  
3661           • A compressed directory (zip, gzip, etc)  
3662           • A JAR file (or its variants – WAR, EAR, etc)

3663           Contributions do not contain other contributions. If the packaging format is a JAR file that  
3664 contains other JAR files (or any similar nesting of other technologies), the internal files are not  
3665 treated as separate SCA contributions. It is up to the implementation to determine whether the  
3666 internal JAR file should be represented as a single artifact in the contribution hierarchy or whether  
3667 all of the contents should be represented as separate artifacts.

3668           A goal of SCA's approach to deployment is that the contents of a contribution should not need to  
3669 be modified in order to install and use the contents of the contribution in a domain.  
3670

## 3671 12.2.1 SCA Artifact Resolution

3672           Contributions can be self-contained, in that all of the artifacts necessary to run the contents of the  
3673 contribution are found within the contribution itself. However, it can also be the case that the  
3674 contents of the contribution make one or many references to artifacts that are not contained  
3675 within the contribution. These references can be to SCA artifacts such as composites or they can  
3676 be to other artifacts such as WSDL files, XSD files or to code artifacts such as Java class files and  
3677 BPEL process files. Note: This form of artifact resolution does not apply to imports of composite  
3678 files, as described in Section 6.6.

3679           A contribution can use some artifact-related or packaging-related means to resolve artifact  
3680 references. Examples of such mechanisms include:

- 3681           • wsdlLocation and schemaLocation attributes in references to WSDL and XSD schema  
3682 artifacts respectively  
3683           • OSGi bundle mechanisms for resolving Java class and related resource dependencies

3684           Where present, these mechanisms MUST be used by the SCA runtime to resolve artifact  
3685 dependencies. [ASM12005] The SCA runtime MUST raise an error if an artifact cannot be resolved  
3686 using these mechanisms, if present. [ASM12021]  
3687

3688 SCA also provides an artifact resolution mechanism. The SCA artifact resolution mechanism is  
3689 used either where no other mechanisms are available, for example in cases where the  
3690 mechanisms used by the various contributions in the same SCA Domain are different. An example  
3691 of the latter case is where an OSGi Bundle is used for one contribution but where a second  
3692 contribution used by the first one is not implemented using OSGi - eg the second contribution  
3693 relates to a mainframe COBOL service whose interfaces are declared using a WSDL which must be  
3694 accessed by the first contribution.

3695 The SCA artifact resolution is likely to be most useful for SCA domains containing heterogeneous  
3696 mixtures of contribution, where artifact-related or packaging-related mechanisms are unlikely to  
3697 work across different kinds of contribution.

3698 SCA artifact resolution works on the principle that a contribution which needs to use artifacts  
3699 defined elsewhere expresses these dependencies using **import** statements in metadata belonging  
3700 to the contribution. A contribution controls which artifacts it makes available to other  
3701 contributions through **export** statements in metadata attached to the contribution. SCA artifact  
3702 resolution is a general mechanism that can be extended for the handling of specific types of  
3703 artifact. The general mechanism that is described in the following paragraphs is mainly intended  
3704 for the handling of XML artifacts. Other types of artifacts, for example Java classes, use an  
3705 extended version of artifact resolution that is specialized to their nature (eg. instead of  
3706 "namespaces", Java uses "packages"). Descriptions of these more specialized forms of artifact  
3707 resolution are contained in the SCA specifications that deal with those artifact types.

3708 Import and export statements for XML artifacts work at the level of namespaces - so that an  
3709 import statement declares that artifacts from a specified namespace are found in other  
3710 contributions, while an export statement makes all the artifacts from a specified namespace  
3711 available to other contributions.

3712 An import declaration can simply specify the namespace to import. In this case, the locations  
3713 which are searched for artifacts in that namespace are the contribution(s) in the Domain which  
3714 have export declarations for the same namespace, if any. Alternatively an import declaration can  
3715 specify a location from which artifacts for the namespace are obtained, in which case, that specific  
3716 location is searched. There can be multiple import declarations for a given namespace. Where  
3717 multiple import declarations are made for the same namespace, all the locations specified MUST  
3718 be searched in lexical order. [ASM12022]

3719 For an XML namespace, artifacts can be declared in multiple locations - for example a given  
3720 namespace can have a WSDL declared in one contribution and have an XSD defining XML data  
3721 types in a second contribution.

3722 If the same artifact is declared in multiple locations, this is not an error. The first location as  
3723 defined by lexical order is chosen. If no locations are specified no order exists and the one chosen  
3724 is implementation dependent.

3725 When a contribution contains a reference to an artifact from a namespace that is declared in an import  
3726 statement of the contribution, if the SCA artifact resolution mechanism is used to resolve the artifact, the  
3727 SCA runtime MUST resolve artifacts in the following order:

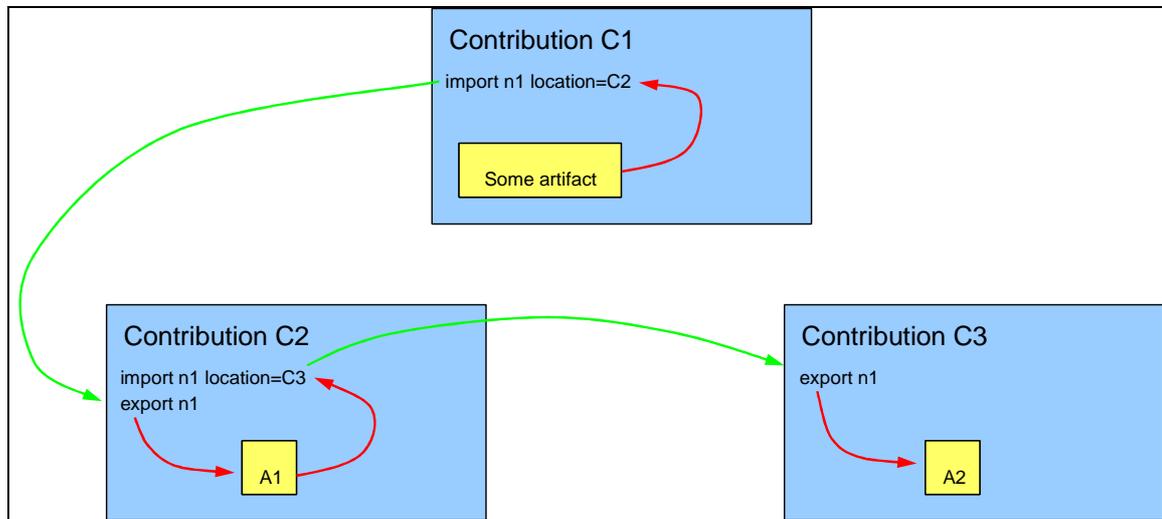
- 3728 1. from the locations identified by the import statement(s) for the namespace. Locations MUST NOT  
3729 be searched recursively in order to locate artifacts (ie only a one-level search is performed).
- 3730 2. from the contents of the contribution itself. [ASM12023]

3731 When a contribution uses an artifact contained in another contribution through SCA artifact  
3732 resolution, if that artifact itself has dependencies on other artifacts, the SCA runtime MUST resolve  
3733 these dependencies in the context of the contribution containing the artifact, not in the context of  
3734 the original contribution. [ASM12024]

3735 For example:

- 3736 • a first contribution "C1" references an artifact "A1" in the namespace "n1" and imports the  
3737 "n1" namespace from a second contribution "C2".
- 3738 • in contribution "C2" the artifact "A1" in the "n1" namespace references an artifact "A2"  
3739 also in the "n1" namespace", which is resolved through an import of the "n1" namespace  
3740 in "C2" which specifies the location "C3".

3741



3742

3743 The "A2" artifact is contained within the third contribution "C3" from which it is resolved by the  
3744 contribution "C2". The "C3" contribution is never used to resolve artifacts directly for the "C1"  
3745 contribution, since "C3" is not declared as an import location for "C1".

3746 For example, if for a contribution "C1", an import is used to resolve a composite "X1" contained in  
3747 contribution "C2", and composite "X1" contains references to other artifacts such as WSDL files or  
3748 XSDs, those references in "X1" are resolved in the context of contribution "C2" and not in the  
3749 context of contribution "C1".

3750 The SCA runtime MUST ignore local definitions of an artifact if the artifact is found through  
3751 resolving an import statement. [ASM12024]

3752 The SCA runtime MUST raise an error if an artifact cannot be resolved by the precedence order  
3753 above. [ASM12025]

3754

## 3755 12.2.2 SCA Contribution Metadata Document

3756 The contribution optionally contains a document that declares runnable composites, exported  
3757 definitions and imported definitions. The document is found at the path of META-INF/sca-  
3758 contribution.xml relative to the root of the contribution. Frequently some SCA metadata needs to  
3759 be specified by hand while other metadata is generated by tools (such as the <import> elements  
3760 described below). To accommodate this, it is also possible to have an identically structured  
3761 document at META-INF/sca-contribution-generated.xml. If this document exists (or is generated  
3762 on an as-needed basis), it will be merged into the contents of sca-contribution.xml, with the  
3763 entries in sca-contribution.xml taking priority if there are any conflicting declarations.  
3764

3765 The format of the document is:

```

3766 <?xml version="1.0" encoding="ASCII"?>
3767 <!-- sca-contribution pseudo-schema -->
3768 <contribution xmlns=http://docs.oasis-open.org/ns/opencsa/sca/200712>

```

3769

Figure 14: Example of SCA Artifact Resolution between Contributions

```
3770 <deployable composite="xs:QName"/>*
3771 <import namespace="xs:String" location="xs:AnyURI"?/>*
3772 <export namespace="xs:String"/>*
3773
3774 </contribution>
3775
```

3776 **deployable element:** Identifies a composite which is a composite within the contribution that is a  
3777 composite intended for potential inclusion into the virtual domain-level composite. Other  
3778 composites in the contribution are not intended for inclusion but only for use by other composites.  
3779 New composites can be created for a contribution after it is installed, by using the [add Deployment](#)  
3780 [Composite](#) capability and the add To Domain Level Composite capability.

3781 Attributes of the deployable element:

- 3782 • **composite (1..1)** – The QName of a composite within the contribution.

3783

3784 **Export element:** A declaration that artifacts belonging to a particular namespace are exported  
3785 and are available for use within other contributions. An export declaration in a contribution  
3786 specifies a namespace, all of whose definitions are considered to be exported. By default,  
3787 definitions are not exported.

3788 The SCA artifact export is useful for SCA domains containing heterogeneous mixtures of  
3789 contribution packagings and technologies, where artifact-related or packaging-related mechanisms  
3790 are unlikely to work across different kinds of contribution.

3791 Attributes of the export element:

- 3792 • **namespace (1..1)** – For XML definitions, which are identified by QNames, the namespace  
3793 should be the namespace URI for the exported definitions. For XML technologies that  
3794 define multiple *symbol spaces* that can be used within one namespace (e.g. WSDL port  
3795 types are a different symbol space from WSDL bindings), all definitions from all symbol  
3796 spaces are exported.

3797  
3798 Technologies that use naming schemes other than QNames must use a different export  
3799 element from the same substitution group as the the SCA <export> element. The  
3800 element used identifies the technology, and can use any value for the namespace that is  
3801 appropriate for that technology. For example, <export.java> can be used can be used to  
3802 export java definitions, in which case the namespace is a fully qualified package name.

3803

3804 **Import element:** Import declarations specify namespaces of definitions that are needed by the  
3805 definitions and implementations within the contribution, but which are not present in the  
3806 contribution. It is expected that in most cases import declarations will be generated based on  
3807 introspection of the contents of the contribution. In this case, the import declarations would be  
3808 found in the META-INF/ sca-contribution-generated.xml document.

3809 Attributes of the import element:

- 3810 • **namespace (1..1)** – For XML definitions, which are identified by QNames, the namespace  
3811 is the namespace URI for the imported definitions. For XML technologies that define  
3812 multiple *symbol spaces* that can be used within one namespace (e.g. WSDL port types are  
3813 a different symbol space from WSDL bindings), all definitions from all symbol spaces are  
3814 imported.

3815  
3816 Technologies that use naming schemes other than QNames must use a different import  
3817 element from the same substitution group as the the SCA <import> element. The  
3818 element used identifies the technology, and can use any value for the namespace that is  
3819 appropriate for that technology. For example, <import.java> can be used can be used to  
3820 import java definitions, in which case the namespace is a fully qualified package name.

3821           • **location (0..1)** – a URI to resolve the definitions for this import. SCA makes no specific  
3822 requirements for the form of this URI, nor the means by which it is resolved. It can point  
3823 to another contribution (through its URI) or it can point to some location entirely outside  
3824 the SCA Domain.  
3825

3826 It is expected that SCA runtimes can define implementation specific ways of resolving location  
3827 information for artifact resolution between contributions. These mechanisms will however usually  
3828 be limited to sets of contributions of one runtime technology and one hosting environment.

3829 In order to accommodate imports of artifacts between contributions of disparate runtime  
3830 technologies, it is strongly suggested that SCA runtimes honor SCA contribution URIs as location  
3831 specification.

3832 SCA runtimes that support contribution URIs for cross-contribution resolution of SCA artifacts are  
3833 expected to do so similarly when used as @schemaLocation and @wsdlLocation and other artifact  
3834 location specifications.

3835 The order in which the import statements are specified can play a role in this mechanism. Since  
3836 definitions of one namespace can be distributed across several artifacts, multiple import  
3837 declarations can be made for one namespace.  
3838

3839 The location value is only a default, and dependent contributions listed in the call to  
3840 installContribution can override the value if there is a conflict. However, the specific mechanism  
3841 for resolving conflicts between contributions that define conflicting definitions is implementation  
3842 specific.

3843  
3844 If the value of the location attribute is an SCA contribution URI, then the contribution packaging  
3845 can become dependent on the deployment environment. In order to avoid such a dependency,  
3846 dependent contributions should be specified only when deploying or updating contributions as  
3847 specified in the section 'Operations for Contributions' below.

### 3848 **12.2.3 Contribution Packaging using ZIP**

3849 SCA allows many different packaging formats that SCA runtimes can support, but SCA requires  
3850 that all runtimes MUST support the ZIP packaging format for contributions. [ASM12006] This  
3851 format allows that metadata specified by the section 'SCA Contribution Metadata Document' be  
3852 present. Specifically, it can contain a top-level "META-INF" directory and a "META-INF/sca-  
3853 contribution.xml" file and there can also be an optional "META-INF/sca-contribution-  
3854 generated.xml" file in the package. SCA defined artifacts as well as non-SCA defined artifacts such  
3855 as object files, WSDL definition, Java classes can be present anywhere in the ZIP archive,

3856 A up to date definition of the ZIP file format is published by PKWARE in [an Application Note on the](#)  
3857 [.ZIP file format \[12\]](#).

3858

## 3859 **12.3 Installed Contribution**

3860 As noted in the section above, the contents of a contribution do not need to be modified in order  
3861 to install and use it within a domain. An *installed contribution* is a contribution with all of the  
3862 associated information necessary in order to execute *deployable composites* within the  
3863 contribution.

3864 An installed contribution is made up of the following things:

- 3865           • Contribution Packaging – the contribution that will be used as the starting point for  
3866 resolving all references
- 3867           • Contribution base URI
- 3868           • Dependent contributions: a set of snapshots of other contributions that are used to resolve  
3869 the import statements from the root composite and from other dependent contributions

- 3870                   ○ Dependent contributions might or might not be shared with other installed  
3871                   contributions.
- 3872                   ○ When the snapshot of any contribution is taken is implementation defined, ranging  
3873                   from the time the contribution is installed to the time of execution
- 3874                   • Deployment-time composites.  
3875                   These are composites that are added into an installed contribution after it has been  
3876                   deployed. This makes it possible to provide final configuration and access to  
3877                   implementations within a contribution without having to modify the contribution. These  
3878                   are optional, as composites that already exist within the contribution can also be used for  
3879                   deployment.

3880

3881                   Installed contributions provide a context in which to resolve qualified names (e.g. QNames in XML,  
3882                   fully qualified class names in Java).

3883                   If multiple dependent contributions have exported definitions with conflicting qualified names, the  
3884                   algorithm used to determine the qualified name to use is implementation dependent.  
3885                   Implementations of SCA MAY also generate an error if there are conflicting names exported from  
3886                   multiple contributions. [\[ASM12007\]](#)

3887

### 3888   **12.3.1 Installed Artifact URIs**

3889                   When a contribution is installed, all artifacts within the contribution are assigned URIs, which are  
3890                   constructed by starting with the base URI of the contribution and adding the relative URI of each  
3891                   artifact (recalling that SCA requires that any packaging format be able to offer up its artifacts in a  
3892                   single hierarchy).

3893

## 3894   **12.4 Operations for Contributions**

3895                   SCA Domains provide the following conceptual functionality associated with contributions  
3896                   (meaning the function might not be represented as addressable services and also meaning that  
3897                   equivalent functionality might be provided in other ways). The functionality is optional meaning  
3898                   that some SCA runtimes MAY choose not to provide the contribution functions functionality in any  
3899                   way. [\[ASM12008\]](#)

### 3900   **12.4.1 install Contribution & update Contribution**

3901                   Creates or updates an installed contribution with a supplied root contribution, and installed at a  
3902                   supplied base URI. A supplied dependent contribution list (<export/> elements) specifies the  
3903                   contributions that should be used to resolve the dependencies of the root contribution and other  
3904                   dependent contributions. These override any dependent contributions explicitly listed via the  
3905                   location attribute in the import statements of the contribution.

3906                   SCA follows the simplifying assumption that the use of a contribution for resolving anything also  
3907                   means that all other exported artifacts can be used from that contribution. Because of this, the  
3908                   dependent contribution list is just a list of installed contribution URIs. There is no need to specify  
3909                   what is being used from each one.

3910                   Each dependent contribution is also an installed contribution, with its own dependent  
3911                   contributions. By default these dependent contributions of the dependent contributions (which we  
3912                   will call *indirect dependent contributions*) are included as dependent contributions of the installed  
3913                   contribution. However, if a contribution in the dependent contribution list exports any conflicting  
3914                   definitions with an indirect dependent contribution, then the indirect dependent contribution is not  
3915                   included (i.e. the explicit list overrides the default inclusion of indirect dependent contributions).  
3916                   Also, if there is ever a conflict between two indirect dependent contributions, then the conflict  
3917                   MUST be resolved by an explicit entry in the dependent contribution list. [\[ASM12009\]](#)

3918 Note that in many cases, the dependent contribution list can be generated. In particular, if the  
3919 creator of a domain is careful to avoid creating duplicate definitions for the same qualified name,  
3920 then it is easy for this list to be generated by tooling.

## 3921 12.4.2 add Deployment Composite & update Deployment Composite

3922 Adds or updates a deployment composite using a supplied composite ("composite by value" – a  
3923 data structure, not an existing resource in the domain) to the contribution identified by a supplied  
3924 contribution URI. The added or updated deployment composite is given a relative URI that  
3925 matches the @name attribute of the composite, with a ".composite" suffix. Since all composites  
3926 must run within the context of a installed contribution (any component implementations or other  
3927 definitions are resolved within that contribution), this functionality makes it possible for the  
3928 deployer to create a composite with final configuration and wiring decisions and add it to an  
3929 installed contribution without having to modify the contents of the root contribution.

3930 Also, in some use cases, a contribution might include only implementation code (e.g. PHP scripts).  
3931 It is then possible for those to be given component names by a (possibly generated) composite  
3932 that is added into the installed contribution, without having to modify the packaging.

## 3933 12.4.3 remove Contribution

3934 Removes the deployed contribution identified by a supplied contribution URI.

3935

## 3936 12.5 Use of Existing (non-SCA) Mechanisms for Resolving Artifacts

3937

3938 For certain types of artifact, there are existing and commonly used mechanisms for referencing a  
3939 specific concrete location where the artifact can be resolved.

3940 Examples of these mechanisms include:

- 3941 • For WSDL files, the **@wsdlLocation** attribute is a hint that has a URI value pointing to the  
3942 place holding the WSDL itself.
- 3943 • For XSDs, the **@schemaLocation** attribute is a hint which matches the namespace to a  
3944 URI where the XSD is found.

3945 **Note:** In neither of these cases is the runtime obliged to use the location hint and the URI does  
3946 not have to be dereferenced.

3947 SCA permits the use of these mechanisms. Where present, non-SCA artifact resolution  
3948 mechanisms MUST be used by the SCA runtime in precedence to the SCA mechanisms.  
3949 [ASM12010] However, use of these mechanisms is discouraged because tying assemblies to  
3950 addresses in this way makes the assemblies less flexible and prone to errors when changes are  
3951 made to the overall SCA Domain.

3952 **Note:** If one of the non-SCA artifact resolution mechanisms is present, but there is a failure to  
3953 find the resource indicated when using the mechanism (eg the URI is incorrect or invalid, say) the  
3954 SCA runtime MUST raise an error and MUST NOT attempt to use SCA resolution mechanisms as an  
3955 alternative. [ASM12011]

3956

## 3957 12.6 Domain-Level Composite

3958 The domain-level composite is a virtual composite, in that it is not defined by a composite  
3959 definition document. Rather, it is built up and modified through operations on the domain.  
3960 However, in other respects it is very much like a composite, since it contains components, wires,  
3961 services and references.

3962

3963 The value of @autowire for the logical domain composite MUST be autowire="false". [ASM12012]

3964

3965 For components at the Domain level, with References for which @autowire="true" applies, the  
3966 behaviour of the SCA runtime for a given Domain MUST take ONE of the 3 following forms:

3967 1) The SCA runtime MAY disallow deployment of any components with autowire References. In  
3968 this case, the SCA runtime MUST generate an exception at the point where the component is  
3969 deployed.

3970 2) The SCA runtime MAY evaluate the target(s) for the reference at the time that the component  
3971 is deployed and not update those targets when later deployment actions occur.

3972 3) The SCA runtime MAY re-evaluate the target(s) for the reference dynamically as later  
3973 deployment actions occur resulting in updated reference targets which match the new Domain  
3974 configuration. How the new configuration of the reference takes place is described by the relevant  
3975 client and implementation specifications.

3976 [ASM12013]

3977 The abstract domain-level functionality for modifying the domain-level composite is as follows,  
3978 although a runtime may supply equivalent functionality in a different form:

### 3979 12.6.1 add To Domain-Level Composite

3980 This functionality adds the composite identified by a supplied URI to the Domain Level Composite.  
3981 The supplied composite URI must refer to a composite within a installed contribution. The  
3982 composite's installed contribution determines how the composite's artifacts are resolved (directly  
3983 and indirectly). The supplied composite is added to the domain composite with semantics that  
3984 correspond to the domain-level composite having an <include> statement that references the  
3985 supplied composite. All of the composite's components become *top-level* components and the  
3986 services become externally visible services (eg. they would be present in a WSDL description of  
3987 the domain).

### 3988 12.6.2 remove From Domain-Level Composite

3989 Removes from the Domain Level composite the elements corresponding to the composite  
3990 identified by a supplied composite URI. This means that the removal of the components, wires,  
3991 services and references originally added to the domain level composite by the identified  
3992 composite.

### 3993 12.6.3 get Domain-Level Composite

3994 Returns a <composite> definition that has an <include> line for each composite that had been  
3995 added to the domain level composite. It is important to note that, in dereferencing the included  
3996 composites, any referenced artifacts must be resolved in terms of that installed composite.

### 3997 12.6.4 get QName Definition

3998 In order to make sense of the domain-level composite (as returned by get Domain-Level  
3999 Composite), it must be possible to get the definitions for named artifacts in the included  
4000 composites. This functionality takes the supplied URI of an installed contribution (which provides  
4001 the context), a supplied qualified name of a definition to look up, and a supplied symbol space (as  
4002 a QName, eg wsdl:PortType). The result is a single definition, in whatever form is appropriate for  
4003 that definition type.

4004 Note that this, like all the other domain-level operations, is a conceptual operation. Its capabilities  
4005 should exist in some form, but not necessarily as a service operation with exactly this signature.

## 4006 12.7 Dynamic Behaviour of Wires in the SCA Domain

4007 For components with references which are at the Domain level, there is the potential for dynamic  
4008 behaviour when the wires for a component reference change (this can only apply to component  
4009 references at the Domain level and not to components within composites used as implementations):

4010 The configuration of the wires for a component reference of a component at the Domain level can change  
4011 by means of deployment actions:

- 4012 1. <wire/> elements can be added, removed or replaced by deployment actions
- 4013 2. Components can be updated by deployment actions (ie this may change the component reference  
4014 configuration)
- 4015 3. Components which are the targets of reference wires can be updated or removed
- 4016 4. Components can be added that are potential targets for references which are marked with  
4017 @autowire=true

4018  
4019 Where <wire/> elements are added, removed or replaced by deployment actions, the components whose  
4020 references are affected by those deployment actions MAY have their references updated by the SCA  
4021 runtime dynamically without the need to stop and start those components. [ASM12014]

4022 Where components are updated by deployment actions (their configuration is changed in some way,  
4023 which may include changing the wires of component references), the new configuration MUST apply to all  
4024 new instances of those components once the update is complete. [ASM12015] An SCA runtime MAY  
4025 choose to maintain existing instances with the old configuration of components updated by deployment  
4026 actions, but an SCA runtime MAY choose to stop and discard existing instances of those components.  
4027 [ASM12016]

4028 Where a component that is the target of a wire is removed, without the wire being changed, then future  
4029 invocations of the reference that use that wire SHOULD fail with a ServiceUnavailable fault. If the wire is  
4030 the result of the autowire process, the SCA runtime MUST:

- 4031 • either cause future invocation of the target component's services to fail with a  
4032 ServiceUnavailable fault
- 4033 • or alternatively, if an alternative target component is available that satisfies the autowire  
4034 process, update the reference of the source component [ASM12017]

4035 Where a component that is the target of a wire is updated, future invocations of that reference SHOULD  
4036 use the updated component. [ASM12018] Where an existing domain level component is updated, an  
4037 SCA runtime MAY maintain a copy of a component offering a conversational service until all existing  
4038 conversations complete - alternatively all existing conversations MAY be terminated. [ASM12019]

4039 Where a component is added to the domain that is a potential target for a domain level component  
4040 reference where that reference is marked as @autowire=true, the SCA runtime MUST:

- 4041 - either update the references for the source component once the new component is running.
- 4042 - or alternatively, defer the updating of the references of the source component until the source  
4043 component is stopped and restarted. [ASM12020]

4044

## 4045 12.8 Dynamic Behaviour of Component Property Values

4046 For a domain level component with a Property whose value is obtained from a Domain-level Property  
4047 through the use of the @source attribute, if the domain level property is updated by means of deployment  
4048 actions, the SCA runtime MUST

- 4049 - either update the property value of the domain level component. once the update of the domain  
4050 property is complete
- 4051 - or alternative defer the updating of the component property value until the component is stopped  
4052 and restarted

---

4053

## 13 Conformance

4054

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

4055

4056

An SCA runtime MUST reject a composite file that does not conform to the sca-core.xsd schema..

4057

[ASM13001]

4058

4059

## A. XML Schemas

4060

### A.1 sca.xsd

4061

```
4062 <?xml version="1.0" encoding="UTF-8"?>
4063 <!-- Copyright(C) OASIS(R) 2005,2008. All Rights Reserved. OASIS trademark,
4064 IPR and other policies apply. -->
4065 <schema xmlns="http://www.w3.org/2001/XMLSchema"
4066 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4067 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712">
4068
4069 <include schemaLocation="sca-core-1.1-schema-200803.xsd"/>
4070
4071 <include schemaLocation="sca-interface-java-1.1-schema-200803.xsd"/>
4072 <include schemaLocation="sca-interface-wsdl-1.1-schema-200803.xsd"/>
4073 <include schemaLocation="sca-interface-cpp-1.1-schema-200803.xsd"/>
4074 <include schemaLocation="sca-interface-c-1.1-schema-200803.xsd"/>
4075
4076 <include schemaLocation="sca-implementation-java-1.1-schema-200803.xsd"/>
4077 <include schemaLocation=
4078 "sca-implementation-composite-1.1-schema-200803.xsd"/>
4079 <include schemaLocation="sca-implementation-cpp-1.1-schema-200803.xsd"/>
4080 <include schemaLocation="sca-implementation-c-1.1-schema-200803.xsd"/>
4081 <include schemaLocation="sca-implementation-bpel-1.1-schema-200803.xsd"/>
4082
4083 <include schemaLocation="sca-binding-webservice-1.1-schema-200803.xsd"/>
4084 <include schemaLocation="sca-binding-jms-1.1-schema-200803.xsd"/>
4085 <include schemaLocation="sca-binding-sca-1.1-schema-200803.xsd"/>
4086
4087 <include schemaLocation="sca-definitions-1.1-schema-200803.xsd"/>
4088 <include schemaLocation="sca-policy-1.1-schema-200803.xsd"/>
4089
4090 <include schemaLocation="sca-contribution-1.1-schema-200803.xsd"/>
4091
4092 </schema>
```

4093

### A.2 sca-core.xsd

4094

```
4095 <?xml version="1.0" encoding="UTF-8"?>
4096 <!-- Copyright(C) OASIS(R) 2005,2008. All Rights Reserved. OASIS trademark,
4097 IPR and other policies apply. -->
4098 <schema xmlns="http://www.w3.org/2001/XMLSchema"
4099 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4100 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4101 elementFormDefault="qualified">
4102
4103 <import namespace="http://www.w3.org/XML/1998/namespace"
4104 schemaLocation="http://www.w3.org/2001/xml.xsd"/>
4105
4106 <!-- Common extension base for SCA definitions -->
4107 <complexType name="CommonExtensionBase">
4108 <sequence>
```

```

4109 <element ref="sca:documentation" minOccurs="0"
4110 maxOccurs="unbounded"/>
4111 </sequence>
4112 <anyAttribute namespace="##other" processContents="lax"/>
4113 </complexType>
4114
4115 <element name="documentation" type="sca:Documentation"/>
4116 <complexType name="Documentation" mixed="true">
4117 <sequence>
4118 <any namespace="##other" processContents="lax" minOccurs="0"
4119 maxOccurs="unbounded"/>
4120 </sequence>
4121 <attribute ref="xml:lang"/>
4122 </complexType>
4123
4124 <!-- Component Type -->
4125 <element name="componentType" type="sca:ComponentType"/>
4126 <complexType name="ComponentType">
4127 <complexContent>
4128 <extension base="sca:CommonExtensionBase">
4129 <sequence>
4130 <element ref="sca:implementation" minOccurs="0"/>
4131 <choice minOccurs="0" maxOccurs="unbounded">
4132 <element name="service" type="sca:ComponentService"/>
4133 <element name="reference"
4134 type="sca:ComponentTypeReference"/>
4135 <element name="property" type="sca:Property"/>
4136 </choice>
4137 <any namespace="##other" processContents="lax" minOccurs="0"
4138 maxOccurs="unbounded"/>
4139 </sequence>
4140 <attribute name="constrainingType" type="QName" use="optional"/>
4141 </extension>
4142 </complexContent>
4143 </complexType>
4144
4145 <!-- Composite -->
4146 <element name="composite" type="sca:Composite"/>
4147 <complexType name="Composite">
4148 <complexContent>
4149 <extension base="sca:CommonExtensionBase">
4150 <sequence>
4151 <element name="include" type="anyURI" minOccurs="0"
4152 maxOccurs="unbounded"/>
4153 <choice minOccurs="0" maxOccurs="unbounded">
4154 <element name="service" type="sca:Service"/>
4155 <element name="property" type="sca:Property"/>
4156 <element name="component" type="sca:Component"/>
4157 <element name="reference" type="sca:Reference"/>
4158 <element name="wire" type="sca:Wire"/>
4159 </choice>
4160 <any namespace="##other" processContents="lax" minOccurs="0"
4161 maxOccurs="unbounded"/>
4162 </sequence>
4163 <attribute name="name" type="NCName" use="required"/>
4164 <attribute name="targetNamespace" type="anyURI" use="required"/>
4165 <attribute name="local" type="boolean" use="optional"
4166 default="false"/>

```

```

4167 <attribute name="autowire" type="boolean" use="optional"
4168 default="false"/>
4169 <attribute name="constrainingType" type="QName" use="optional"/>
4170 <attribute name="requires" type="sca:listOfQNames"
4171 use="optional"/>
4172 <attribute name="policySets" type="sca:listOfQNames"
4173 use="optional"/>
4174 </extension>
4175 </complexContent>
4176 </complexType>
4177
4178 <!-- Contract base type for Service, Reference -->
4179 <complexType name="Contract" abstract="true">
4180 <complexContent>
4181 <extension base="sca:CommonExtensionBase">
4182 <sequence>
4183 <element ref="sca:interface" minOccurs="0" maxOccurs="1" />
4184 <element name="operation" type="sca:Operation" minOccurs="0"
4185 maxOccurs="unbounded" />
4186 <element ref="sca:binding" minOccurs="0"
4187 maxOccurs="unbounded"/>
4188 <element ref="sca:callback" minOccurs="0" maxOccurs="1" />
4189 <any namespace="##other" processContents="lax" minOccurs="0"
4190 maxOccurs="unbounded" />
4191 </sequence>
4192 <attribute name="name" type="NCName" use="required" />
4193 <attribute name="requires" type="sca:listOfQNames"
4194 use="optional"/>
4195 <attribute name="policySets" type="sca:listOfQNames"
4196 use="optional"/>
4197 </extension>
4198 </complexContent>
4199 </complexType>
4200
4201 <!-- Service -->
4202 <complexType name="Service">
4203 <complexContent>
4204 <extension base="sca:Contract">
4205 <attribute name="promote" type="anyURI" use="required"/>
4206 </extension>
4207 </complexContent>
4208 </complexType>
4209
4210 <!-- Interface -->
4211 <element name="interface" type="sca:Interface" abstract="true"/>
4212 <complexType name="Interface" abstract="true">
4213 <complexContent>
4214 <extension base="sca:CommonExtensionBase"/>
4215 </complexContent>
4216 </complexType>
4217
4218 <!-- Reference -->
4219 <complexType name="Reference">
4220 <complexContent>
4221 <extension base="sca:Contract">
4222 <attribute name="autowire" type="boolean" use="optional"/>
4223 <attribute name="target" type="sca:listOfAnyURIs"
4224 use="optional"/>

```

```

4225 <attribute name="wiredByImpl" type="boolean" use="optional"
4226 default="false"/>
4227 <attribute name="multiplicity" type="sca:Multiplicity"
4228 use="optional" default="1..1"/>
4229 <attribute name="promote" type="sca:listOfAnyURIs"
4230 use="required"/>
4231 </extension>
4232 </complexContent>
4233 </complexType>
4234
4235 <!-- Property -->
4236 <complexType name="SCAPropertyBase" mixed="true">
4237 <sequence>
4238 <any namespace="##any" processContents="lax" minOccurs="0"/>
4239 <!-- NOT an extension point; This any exists to accept
4240 the element-based or complex type property
4241 i.e. no element-based extension point under "sca:property" -->
4242 </sequence>
4243 <!-- mixed="true" to handle simple type -->
4244 <attribute name="requires" type="sca:listOfQNames" use="optional"/>
4245 <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
4246 </complexType>
4247
4248 <complexType name="Property" mixed="true">
4249 <complexContent mixed="true">
4250 <extension base="sca:SCAPropertyBase">
4251 <attribute name="name" type="NCName" use="required"/>
4252 <attribute name="type" type="QName" use="optional"/>
4253 <attribute name="element" type="QName" use="optional"/>
4254 <attribute name="many" type="boolean" use="optional"
4255 default="false"/>
4256 <attribute name="mustSupply" type="boolean" use="optional"
4257 default="false"/>
4258 <anyAttribute namespace="##any" processContents="lax"/>
4259 </extension>
4260 <!-- extension defines the place to hold default value -->
4261 <!-- an extension point ; attribute-based only -->
4262 </complexContent>
4263 </complexType>
4264
4265 <complexType name="PropertyValue" mixed="true">
4266 <complexContent mixed="true">
4267 <extension base="sca:SCAPropertyBase">
4268 <attribute name="name" type="NCName" use="required"/>
4269 <attribute name="type" type="QName" use="optional"/>
4270 <attribute name="element" type="QName" use="optional"/>
4271 <attribute name="many" type="boolean" use="optional"
4272 default="false"/>
4273 <attribute name="source" type="string" use="optional"/>
4274 <attribute name="file" type="anyURI" use="optional"/>
4275 <anyAttribute namespace="##any" processContents="lax"/>
4276 </extension>
4277 <!-- an extension point ; attribute-based only -->
4278 </complexContent>
4279 </complexType>
4280
4281 <!-- Binding -->
4282 <element name="binding" type="sca:Binding" abstract="true"/>

```

```

4283 <complexType name="Binding" abstract="true">
4284 <complexContent>
4285 <extension base="sca:CommonExtensionBase">
4286 <sequence>
4287 <element ref="sca:wireFormat" minOccurs="0" maxOccurs="1" />
4288 <element ref="sca:operationSelector"
4289 minOccurs="0" maxOccurs="1" />
4290 <element name="operation" type="sca:Operation" minOccurs="0"
4291 maxOccurs="unbounded" />
4292 </sequence>
4293 <attribute name="uri" type="anyURI" use="optional" />
4294 <attribute name="name" type="NCName" use="optional" />
4295 <attribute name="requires" type="sca:listOfQNames"
4296 use="optional" />
4297 <attribute name="policySets" type="sca:listOfQNames"
4298 use="optional" />
4299 </extension>
4300 </complexContent>
4301 </complexType>
4302
4303 <!-- Binding Type -->
4304 <element name="bindingType" type="sca:BindingType" />
4305 <complexType name="BindingType">
4306 <complexContent>
4307 <extension base="sca:CommonExtensionBase">
4308 <sequence>
4309 <any namespace="##other" processContents="lax" minOccurs="0"
4310 maxOccurs="unbounded" />
4311 </sequence>
4312 <attribute name="type" type="QName" use="required" />
4313 <attribute name="alwaysProvides" type="sca:listOfQNames"
4314 use="optional" />
4315 <attribute name="mayProvide" type="sca:listOfQNames"
4316 use="optional" />
4317 </extension>
4318 </complexContent>
4319 </complexType>
4320
4321 <!-- WireFormat Type -->
4322 <element name="wireFormat" type="sca:WireFormatType" />
4323 <complexType name="WireFormatType" abstract="true">
4324 <sequence>
4325 <any namespace="##other" processContents="lax" minOccurs="0"
4326 maxOccurs="unbounded" />
4327 </sequence>
4328 <anyAttribute namespace="##other" processContents="lax" />
4329 </complexType>
4330
4331 <!-- OperationSelector Type -->
4332 <element name="operationSelector" type="sca:OperationSelectorType" />
4333 <complexType name="OperationSelectorType" abstract="true">
4334 <sequence>
4335 <any namespace="##other" processContents="lax" minOccurs="0"
4336 maxOccurs="unbounded" />
4337 </sequence>
4338 <anyAttribute namespace="##other" processContents="lax" />
4339 </complexType>
4340 <!-- Callback -->

```

```

4341 <element name="callback" type="sca:Callback"/>
4342 <complexType name="Callback">
4343 <complexContent>
4344 <extension base="sca:CommonExtensionBase">
4345 <choice minOccurs="0" maxOccurs="unbounded">
4346 <element ref="sca:binding"/>
4347 <any namespace="##other" processContents="lax"/>
4348 </choice>
4349 <attribute name="requires" type="sca:listOfQNames"
4350 use="optional"/>
4351 <attribute name="policySets" type="sca:listOfQNames"
4352 use="optional"/>
4353 </extension>
4354 </complexContent>
4355 </complexType>
4356
4357 <!-- Component -->
4358 <complexType name="Component">
4359 <complexContent>
4360 <extension base="sca:CommonExtensionBase">
4361 <sequence>
4362 <element ref="sca:implementation" minOccurs="0"/>
4363 <choice minOccurs="0" maxOccurs="unbounded">
4364 <element name="service" type="sca:ComponentService"/>
4365 <element name="reference" type="sca:ComponentReference"/>
4366 <element name="property" type="sca:PropertyValue"/>
4367 </choice>
4368 <any namespace="##other" processContents="lax" minOccurs="0"
4369 maxOccurs="unbounded"/>
4370 </sequence>
4371 <attribute name="name" type="NCName" use="required"/>
4372 <attribute name="autowire" type="boolean" use="optional"/>
4373 <attribute name="constrainingType" type="QName" use="optional"/>
4374 <attribute name="requires" type="sca:listOfQNames"
4375 use="optional"/>
4376 <attribute name="policySets" type="sca:listOfQNames"
4377 use="optional"/>
4378 </extension>
4379 </complexContent>
4380 </complexType>
4381
4382 <!-- Component Service -->
4383 <complexType name="ComponentService">
4384 <complexContent>
4385 <extension base="sca:Contract">
4386 </extension>
4387 </complexContent>
4388 </complexType>
4389
4390 <!-- Component Reference -->
4391 <complexType name="ComponentReference">
4392 <complexContent>
4393 <extension base="sca:Contract">
4394 <attribute name="autowire" type="boolean" use="optional"/>
4395 <attribute name="target" type="sca:listOfAnyURIs"
4396 use="optional"/>
4397 <attribute name="wiredByImpl" type="boolean" use="optional"
4398 default="false"/>

```

```

4399 <attribute name="multiplicity" type="sca:Multiplicity"
4400 use="optional" default="1..1"/>
4401 </extension>
4402 </complexContent>
4403 </complexType>
4404
4405 <!-- Component Type Reference -->
4406 <complexType name="ComponentTypeReference">
4407 <complexContent>
4408 <restriction base="sca:ComponentReference">
4409 <sequence>
4410 <element ref="sca:documentation" minOccurs="0"
4411 maxOccurs="unbounded"/>
4412 <element ref="sca:interface" minOccurs="0"/>
4413 <element name="operation" type="sca:Operation" minOccurs="0"
4414 maxOccurs="unbounded"/>
4415 <element ref="sca:binding" minOccurs="0"
4416 maxOccurs="unbounded"/>
4417 <element ref="sca:callback" minOccurs="0"/>
4418 <any namespace="##other" processContents="lax" minOccurs="0"
4419 maxOccurs="unbounded"/>
4420 </sequence>
4421 <attribute name="name" type="NCName" use="required"/>
4422 <attribute name="autowire" type="boolean" use="optional"/>
4423 <attribute name="wiredByImpl" type="boolean" use="optional"
4424 default="false"/>
4425 <attribute name="multiplicity" type="sca:Multiplicity"
4426 use="optional" default="1..1"/>
4427 <attribute name="requires" type="sca:listOfQNames"
4428 use="optional"/>
4429 <attribute name="policySets" type="sca:listOfQNames"
4430 use="optional"/>
4431 <anyAttribute namespace="##other" processContents="lax"/>
4432 </restriction>
4433 </complexContent>
4434 </complexType>
4435
4436 <!-- Implementation -->
4437 <element name="implementation" type="sca:Implementation" abstract="true"/>
4438 <complexType name="Implementation" abstract="true">
4439 <complexContent>
4440 <extension base="sca:CommonExtensionBase">
4441 <attribute name="requires" type="sca:listOfQNames"
4442 use="optional"/>
4443 <attribute name="policySets" type="sca:listOfQNames"
4444 use="optional"/>
4445 </extension>
4446 </complexContent>
4447 </complexType>
4448
4449 <!-- Implementation Type -->
4450 <element name="implementationType" type="sca:ImplementationType"/>
4451 <complexType name="ImplementationType">
4452 <complexContent>
4453 <extension base="sca:CommonExtensionBase">
4454 <sequence>
4455 <any namespace="##other" processContents="lax" minOccurs="0"
4456 maxOccurs="unbounded"/>

```

```

4457 </sequence>
4458 <attribute name="type" type="QName" use="required"/>
4459 <attribute name="alwaysProvides" type="sca:listOfQNames"
4460 use="optional"/>
4461 <attribute name="mayProvide" type="sca:listOfQNames"
4462 use="optional"/>
4463 </extension>
4464 </complexContent>
4465 </complexType>
4466
4467 <!-- Wire -->
4468 <complexType name="Wire">
4469 <complexContent>
4470 <extension base="sca:CommonExtensionBase">
4471 <sequence>
4472 <any namespace="##other" processContents="lax" minOccurs="0"
4473 maxOccurs="unbounded"/>
4474 </sequence>
4475 <attribute name="source" type="anyURI" use="required"/>
4476 <attribute name="target" type="anyURI" use="required"/>
4477 </extension>
4478 </complexContent>
4479 </complexType>
4480
4481 <!-- Include -->
4482 <element name="include" type="sca:Include"/>
4483 <complexType name="Include">
4484 <complexContent>
4485 <extension base="sca:CommonExtensionBase">
4486 <attribute name="name" type="QName"/>
4487 </extension>
4488 </complexContent>
4489 </complexType>
4490
4491 <!-- Operation -->
4492 <complexType name="Operation">
4493 <complexContent>
4494 <extension base="sca:CommonExtensionBase">
4495 <attribute name="name" type="NCName" use="required"/>
4496 <attribute name="requires" type="sca:listOfQNames"
4497 use="optional"/>
4498 <attribute name="policySets" type="sca:listOfQNames"
4499 use="optional"/>
4500 </extension>
4501 </complexContent>
4502 </complexType>
4503
4504 <!-- Constraining Type -->
4505 <element name="constrainingType" type="sca:ConstrainingType"/>
4506 <complexType name="ConstrainingType">
4507 <complexContent>
4508 <extension base="sca:CommonExtensionBase">
4509 <sequence>
4510 <choice minOccurs="0" maxOccurs="unbounded">
4511 <element name="service" type="sca:ComponentService"/>
4512 <element name="reference" type="sca:ComponentReference"/>
4513 <element name="property" type="sca:Property"/>
4514 </choice>

```

```

4515 <any namespace="##other" processContents="lax" minOccurs="0"
4516 maxOccurs="unbounded"/>
4517 </sequence>
4518 <attribute name="name" type="NCName" use="required"/>
4519 <attribute name="targetNamespace" type="anyURI"/>
4520 <attribute name="requires" type="sca:listOfQNames"
4521 use="optional"/>
4522 </extension>
4523 </complexContent>
4524 </complexType>
4525
4526 <!-- Intents within WSDL documents -->
4527 <attribute name="requires" type="sca:listOfQNames"/>
4528
4529 <!-- Marker for operations ending a conversation -->
4530 <attribute name="endsConversation" type="boolean" default="false"/>
4531
4532 <!-- Global attribute definition for @callback to mark a WSDL port type
4533 as having a callback interface defined in terms of a second port
4534 type. -->
4535 <attribute name="callback" type="anyURI"/>
4536
4537 <!-- Miscellaneous simple type definitions -->
4538 <simpleType name="Multiplicity">
4539 <restriction base="string">
4540 <enumeration value="0..1"/>
4541 <enumeration value="1..1"/>
4542 <enumeration value="0..n"/>
4543 <enumeration value="1..n"/>
4544 </restriction>
4545 </simpleType>
4546
4547 <simpleType name="OverrideOptions">
4548 <restriction base="string">
4549 <enumeration value="no"/>
4550 <enumeration value="may"/>
4551 <enumeration value="must"/>
4552 </restriction>
4553 </simpleType>
4554
4555 <simpleType name="listOfQNames">
4556 <list itemType="QName"/>
4557 </simpleType>
4558
4559 <simpleType name="listOfAnyURIs">
4560 <list itemType="anyURI"/>
4561 </simpleType>
4562
4563 </schema>
4564

```

### 4565 **A.3 sca-binding-sca.xsd**

```

4566
4567 <?xml version="1.0" encoding="UTF-8"?>
4568 <!-- Copyright(C) OASIS(R) 2005,2008. All Rights Reserved. OASIS trademark,
4569 IPR and other policies apply. -->
4570 <schema xmlns="http://www.w3.org/2001/XMLSchema"

```

```

4571 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4572 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4573 elementFormDefault="qualified">
4574
4575 <include schemaLocation="sca-core-1.1-schema-200803.xsd" />
4576
4577 <!-- SCA Binding -->
4578 <element name="binding.sca" type="sca:SCABinding"
4579 substitutionGroup="sca:binding"/>
4580 <complexType name="SCABinding">
4581 <complexContent>
4582 <extension base="sca:Binding"/>
4583 </complexContent>
4584 </complexType>
4585
4586 </schema>
4587

```

#### 4588 **A.4 sca-interface-java.xsd**

```

4589
4590 <?xml version="1.0" encoding="UTF-8"?>
4591 <!-- Copyright(C) OASIS(R) 2005,2008. All Rights Reserved. OASIS trademark,
4592 IPR and other policies apply. -->
4593 <schema xmlns="http://www.w3.org/2001/XMLSchema"
4594 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4595 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4596 elementFormDefault="qualified">
4597
4598 <include schemaLocation="sca-core-1.1-schema-200803.xsd" />
4599
4600 <!-- Java Interface -->
4601 <element name="interface.java" type="sca:JavaInterface"
4602 substitutionGroup="sca:interface"/>
4603 <complexType name="JavaInterface">
4604 <complexContent>
4605 <extension base="sca:Interface">
4606 <sequence>
4607 <any namespace="##other" processContents="lax" minOccurs="0"
4608 maxOccurs="unbounded"/>
4609 </sequence>
4610 <attribute name="interface" type="NCName" use="required"/>
4611 <attribute name="callbackInterface" type="NCName"
4612 use="optional"/>
4613 <anyAttribute namespace="##any" processContents="lax"/>
4614 </extension>
4615 </complexContent>
4616 </complexType>
4617
4618 </schema>
4619
4620

```

#### 4621 **A.5 sca-interface-wsdl.xsd**

4622

```

4623 <?xml version="1.0" encoding="UTF-8"?>
4624 <!-- Copyright(C) OASIS(R) 2005,2008. All Rights Reserved. OASIS trademark,
4625 IPR and other policies apply. -->
4626 <schema xmlns="http://www.w3.org/2001/XMLSchema"
4627 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4628 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4629 elementFormDefault="qualified">
4630
4631 <include schemaLocation="sca-core-1.1-schema-200803.xsd"/>
4632
4633 <!-- WSDL Interface -->
4634 <element name="interface.wsdl" type="sca:WSDLPortType"
4635 substitutionGroup="sca:interface"/>
4636 <complexType name="WSDLPortType">
4637 <complexContent>
4638 <extension base="sca:Interface">
4639 <sequence>
4640 <any namespace="##other" processContents="lax" minOccurs="0"
4641 maxOccurs="unbounded"/>
4642 </sequence>
4643 <attribute name="interface" type="anyURI" use="required"/>
4644 <attribute name="callbackInterface" type="anyURI"
4645 use="optional"/>
4646 <anyAttribute namespace="##any" processContents="lax"/>
4647 </extension>
4648 </complexContent>
4649 </complexType>
4650 </schema>
4651
4652
4653

```

## 4654 **A.6 sca-implementation-java.xsd**

```

4655
4656 <?xml version="1.0" encoding="UTF-8"?>
4657 <!-- Copyright(C) OASIS(R) 2005,2008. All Rights Reserved. OASIS trademark,
4658 IPR and other policies apply. -->
4659 <schema xmlns="http://www.w3.org/2001/XMLSchema"
4660 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4661 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4662 elementFormDefault="qualified">
4663
4664 <include schemaLocation="sca-core-1.1-schema-200803.xsd"/>
4665
4666 <!-- Java Implementation -->
4667 <element name="implementation.java" type="sca:JavaImplementation"
4668 substitutionGroup="sca:implementation"/>
4669 <complexType name="JavaImplementation">
4670 <complexContent>
4671 <extension base="sca:Implementation">
4672 <sequence>
4673 <any namespace="##other" processContents="lax" minOccurs="0"
4674 maxOccurs="unbounded"/>
4675 </sequence>
4676 <attribute name="class" type="NCName" use="required"/>
4677 <anyAttribute namespace="##any" processContents="lax"/>

```

```
4678 </extension>
4679 </complexContent>
4680 </complexType>
4681
4682 </schema>
```

## 4683 **A.7 sca-implementation-composite.xsd**

```
4684
4685 <?xml version="1.0" encoding="UTF-8"?>
4686 <!-- Copyright(C) OASIS(R) 2005,2008. All Rights Reserved. OASIS trademark,
4687 IPR and other policies apply. -->
4688 <schema xmlns="http://www.w3.org/2001/XMLSchema"
4689 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4690 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4691 elementFormDefault="qualified">
4692
4693 <include schemaLocation="sca-core-1.1-schema-200803.xsd"/>
4694
4695 <!-- Composite Implementation -->
4696 <element name="implementation.composite" type="sca:SCAImplementation"
4697 substitutionGroup="sca:implementation"/>
4698 <complexType name="SCAImplementation">
4699 <complexContent>
4700 <extension base="sca:Implementation">
4701 <sequence>
4702 <any namespace="##other" processContents="lax" minOccurs="0"
4703 maxOccurs="unbounded"/>
4704 </sequence>
4705 <attribute name="name" type="QName" use="required"/>
4706 </extension>
4707 </complexContent>
4708 </complexType>
4709
4710 </schema>
4711
```

## 4712 **A.8 sca-definitions.xsd**

```
4713
4714 <?xml version="1.0" encoding="UTF-8"?>
4715 <!-- Copyright(C) OASIS(R) 2005,2008. All Rights Reserved. OASIS trademark,
4716 IPR and other policies apply. -->
4717 <schema xmlns="http://www.w3.org/2001/XMLSchema"
4718 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4719 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4720 elementFormDefault="qualified">
4721
4722 <include schemaLocation="sca-core-1.1-schema-200803.xsd"/>
4723 <include schemaLocation="sca-policy-1.1-schema-200803.xsd"/>
4724
4725 <!-- Definitions -->
4726 <element name="definitions" type="sca:tDefinitions"/>
4727 <complexType name="tDefinitions">
4728 <complexContent>
4729 <extension base="sca:CommonExtensionBase">
4730 <choice minOccurs="0" maxOccurs="unbounded">
```

```

4731 <element ref="sca:intent"/>
4732 <element ref="sca:policySet"/>
4733 <element ref="sca:binding"/>
4734 <element ref="sca:bindingType"/>
4735 <element ref="sca:implementationType"/>
4736 <any namespace="##other" processContents="lax" minOccurs="0"
4737 maxOccurs="unbounded"/>
4738 </choice>
4739 </extension>
4740 </complexContent>
4741 </complexType>
4742
4743 </schema>
4744
4745

```

## 4746 **A.9 sca-binding-webservice.xsd**

4747 Is described in [the SCA Web Services Binding specification \[9\]](#)

## 4748 **A.10 sca-binding-jms.xsd**

4749 Is described in [the SCA JMS Binding specification \[11\]](#)

## 4750 **A.11 sca-policy.xsd**

4751 Is described in [the SCA Policy Framework specification \[10\]](#)

## 4753 **A.12 sca-contribution.xsd**

```

4754
4755 <?xml version="1.0" encoding="UTF-8"?>
4756 <!-- Copyright(C) OASIS(R) 2005,2008. All Rights Reserved. OASIS trademark,
4757 IPR and other policies apply. -->
4758 <schema xmlns="http://www.w3.org/2001/XMLSchema"
4759 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4760 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
4761 elementFormDefault="qualified">
4762
4763 <include schemaLocation="sca-core-1.1-schema-200803.xsd"/>
4764
4765 <!-- Contribution -->
4766 <element name="contribution" type="sca:ContributionType"/>
4767 <complexType name="ContributionType">
4768 <complexContent>
4769 <extension base="sca:CommonExtensionBase">
4770 <sequence>
4771 <element name="deployable" type="sca:DeployableType"
4772 maxOccurs="unbounded"/>
4773 <element name="import" type="sca:ImportType" minOccurs="0"
4774 maxOccurs="unbounded"/>
4775 <element name="export" type="sca:ExportType" minOccurs="0"
4776 maxOccurs="unbounded"/>
4777 <any namespace="##other" processContents="lax" minOccurs="0"
4778 maxOccurs="unbounded"/>
4779 </sequence>

```

```

4780 </extension>
4781 </complexContent>
4782 </complexType>
4783
4784 <!-- Deployable -->
4785 <complexType name="DeployableType">
4786 <complexContent>
4787 <extension base="sca:CommonExtensionBase">
4788 <sequence>
4789 <any namespace="##other" processContents="lax" minOccurs="0"
4790 maxOccurs="unbounded"/>
4791 </sequence>
4792 <attribute name="composite" type="QName" use="required"/>
4793 </extension>
4794 </complexContent>
4795 </complexType>
4796
4797 <!-- Import -->
4798 <element name="importBase" type="sca:Import" abstract="true" />
4799 <complexType name="Import" abstract="true">
4800 <complexContent>
4801 <extension base="sca:CommonExtensionBase">
4802 <sequence>
4803 <any namespace="##other" processContents="lax" minOccurs="0"
4804 maxOccurs="unbounded"/>
4805 </sequence>
4806 </extension>
4807 </complexContent>
4808 </complexType>
4809
4810 <element name="import" type="sca:ImportType" />
4811 <complexType name="ImportType">
4812 <complexContent>
4813 <extension base="sca:Import">
4814 <attribute name="namespace" type="string" use="required"/>
4815 <attribute name="location" type="anyURI" use="optional"/>
4816 </extension>
4817 </complexContent>
4818 </complexType>
4819
4820 <!-- Export -->
4821 <element name="exportBase" type="sca:Export" abstract="true" />
4822 <complexType name="Export" abstract="true">
4823 <complexContent>
4824 <extension base="sca:CommonExtensionBase">
4825 <sequence>
4826 <any namespace="##other" processContents="lax" minOccurs="0"
4827 maxOccurs="unbounded"/>
4828 </sequence>
4829 </extension>
4830 </complexContent>
4831 </complexType>
4832
4833 <element name="export" type="sca:ExportType" />
4834 <complexType name="ExportType">
4835 <complexContent>
4836 <extension base="sca:Export">
4837 <attribute name="namespace" type="string" use="required"/>

```

```
4838 </extension>
4839 </complexContent>
4840 </complexType>
4841
4842 </schema>
4843
4844
```

---

4845

## B. SCA Concepts

4846

### B.1 Binding

4847

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

4848

4849

4850

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

4851

4852

4853

4854

### B.2 Component

4855

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

4856

4857

4858

4859

4860

4861

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

4862

4863

4864

4865

4866

### B.3 Service

4867

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

4868

4869

4870

4871

4872

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

4873

4874

4875

4876

4877

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

4878

4879

4880

4881

#### B.3.1 Remotable Service

4882

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

4883

4884

4885

4886

How a Service is identified as remotable is dependant on the Component implementation technology used. See the relevant SCA Implementation Specification for more information. As an example, to define a Remotable Service, a Component implemented in Java would have a Java Interface with the @Remotable annotation

4887

4888

4889

## 4890 B.3.2 Local Service

4891 Local services are services that are designed to be only used “locally” by other implementations that are  
4892 deployed concurrently in a tightly-coupled architecture within the same operating system process.

4893 Local services may rely on by-reference calling conventions, or may assume a very fine-grained  
4894 interaction style that is incompatible with remote distribution. They may also use technology-specific data-  
4895 types.

4896 How a Service is identified as local is dependant on the Component implementation technology used.  
4897 See the relevant SCA Implementation Specification for more information. As an example, to define a  
4898 Local Service, a Component implemented in Java would define a Java Interface that does not have the  
4899 @Remotable annotation.

4900

## 4901 B.4 Reference

4902 **SCA references** represent a dependency that an implementation has on a service that is supplied by  
4903 some other implementation, where the service to be used is specified through configuration. In other  
4904 words, a reference is a service that an implementation may call during the execution of its business  
4905 function. References are typed by an interface.

4906 For composites, composite references can be accessed by components within the composite like any  
4907 service provided by a component within the composite. Composite references can be used as the targets  
4908 of wires from component references when configuring Components.

4909 A composite reference can be used to access a service such as: an SCA service provided by another  
4910 SCA composite, a Web service, a stateless session EJB, a data base stored procedure or an EIS service,  
4911 and so on. References use **bindings** to describe the access method used to their services. SCA provides  
4912 an **extensibility mechanism** that allows the introduction of new binding types to references.

4913

## 4914 B.5 Implementation

4915 An implementation is concept that is used to describe a piece of software technology such as a Java  
4916 class, BPEL process, XSLT transform, or C++ class that is used to implement one or more services in a  
4917 service-oriented application. An SCA composite is also an implementation.

4918 Implementations define points of variability including properties that can be set and settable references to  
4919 other services. The points of variability are configured by a component that uses the implementation. The  
4920 specification refers to the configurable aspects of an implementation as its **componentType**.

## 4921 B.6 Interface

4922 **Interfaces** define one or more business functions. These business functions are provided by Services  
4923 and are used by components through References. Services are defined by the Interface they implement.  
4924 SCA currently supports a number of interface type systems, for example:

- 4925 • Java interfaces
- 4926 • WSDL portTypes
- 4927 • C, C++ header files

4928

4929 SCA also provides an extensibility mechanism by which an SCA runtime can add support for additional  
4930 interface type systems.

4931 Interfaces may be **bi-directional**. A bi-directional service has service operations which must be provided  
4932 by each end of a service communication – this could be the case where a particular service requires a  
4933 “callback” interface on the client, which is calls during the process of handing service requests from the  
4934 client.

4935

## 4936 **B.7 Composite**

4937 An SCA composite is the basic unit of composition within an SCA Domain. An **SCA Composite** is an  
4938 assembly of Components, Services, References, and the Wires that interconnect them. Composites can  
4939 be used to contribute elements to an **SCA Domain**.

4940 A **composite** has the following characteristics:

- 4941 • It may be used as a component implementation. When used in this way, it defines a boundary for  
4942 Component visibility. Components may not be directly referenced from outside of the composite  
4943 in which they are declared.
- 4944 • It can be used to define a unit of deployment. Composites are used to contribute business logic  
4945 artifacts to an SCA domain.

4946

## 4947 **B.8 Composite inclusion**

4948 One composite can be used to provide part of the definition of another composite, through the process of  
4949 inclusion. This is intended to make team development of large composites easier. Included composites  
4950 are merged together into the using composite at deployment time to form a single logical composite.

4951 Composites are included into other composites through `<include.../>` elements in the using composite.  
4952 The SCA Domain uses composites in a similar way, through the deployment of composite files to a  
4953 specific location.

4954

## 4955 **B.9 Property**

4956 **Properties** allow for the configuration of an implementation with externally set data values. The data  
4957 value is provided through a Component, possibly sourced from the property of a containing composite.

4958 Each Property is defined by the implementation. Properties may be defined directly through the  
4959 implementation language or through annotations of implementations, where the implementation language  
4960 permits, or through a componentType file. A Property can be either a simple data type or a complex data  
4961 type. For complex data types, XML schema is the preferred technology for defining the data types.

4962

## 4963 **B.10 Domain**

4964 An SCA Domain represents a set of Services providing an area of Business functionality that is controlled  
4965 by a single organization. As an example, for the accounts department in a business, the SCA Domain  
4966 might cover all finance-related functions, and it might contain a series of composites dealing with specific  
4967 areas of accounting, with one for Customer accounts, another dealing with Accounts Payable.

4968 A domain specifies the instantiation, configuration and connection of a set of components, provided via  
4969 one or more composite files. The domain, like a composite, also has Services and References. Domains  
4970 also contain Wires which connect together the Components, Services and References.

4971

## 4972 **B.11 Wire**

4973 **SCA wires** connect **service references** to **services**.

4974 Valid wire sources are component references. Valid wire targets are component services.

4975 When using included composites, the sources and targets of the wires don't have to be declared in the  
4976 same composite as the composite that contains the wire. The sources and targets can be defined by  
4977 other included composites. Targets can also be external to the SCA domain.

4978

## C. Conformance Items

4979 This section contains a list of conformance items for the SCA Assembly specification.

4980

Conformance ID	Description
[ASM13001]	An SCA runtime MUST reject a composite file that does not conform to the sca-core.xsd schema.
[ASM40001]	The extension of a componentType side file name MUST be .componentType.
[ASM40002]	If present, the @constrainingType attribute of a <componentType/> element MUST reference a <constrainingType/> element in the Domain through its QName.
[ASM40003]	The @name attribute of a <service/> child element of a <componentType/> MUST be unique amongst the service elements of that <componentType/>.
[ASM40004]	The @name attribute of a <reference/> child element of a <componentType/> MUST be unique amongst the reference elements of that <componentType/>.
[ASM40005]	The @name attribute of a <property/> child element of a <componentType/> MUST be unique amongst the property elements of that <componentType/>.
[ASM40006]	If @wiredByImpl is set to "true", then any reference targets configured for this reference MUST be ignored by the runtime.
[ASM40007]	The value of the property @type attribute MUST be the QName of an XML schema type.
[ASM40008]	The value of the property @element attribute MUST be the QName of an XSD global element.
[ASM40009]	The SCA runtime MUST ensure that any implementation default property value is replaced by a value for that property explicitly set by a component using that implementation.
[ASM50001]	The @name attribute of a <component/> child element of a <composite/> MUST be unique amongst the component elements of that <composite/>
[ASM50002]	The @name attribute of a service element of a <component/> MUST be unique amongst the service elements of that <component/>
[ASM50003]	The @name attribute of a service element of a <component/> MUST match the @name attribute of a service element of the componentType of the <implementation/> child element of the component.
[ASM50004]	If a <service/> element has an interface subelement specified, the interface MUST provide a compatible subset of the interface declared on the componentType of the implementation
[ASM50005]	If no binding elements are specified for the service, then the bindings specified for the equivalent service in the componentType of the implementation MUST be used, but if the componentType also has no bindings specified, then <binding.sca/> MUST be used as the binding. If binding elements are specified for the service, then those bindings MUST be used and they override any bindings specified for the equivalent service in the componentType of the implementation.
[ASM50006]	If the callback element is present and contains one or more binding child elements, then those bindings MUST be used for the callback.
[ASM50007]	The @name attribute of a service element of a <component/> MUST be unique amongst the service elements of that <component/>
[ASM50008]	The @name attribute of a reference element of a <component/> MUST match the @name attribute of a reference element of the componentType of the

<implementation/> child element of the component.

- [ASM50009] The value of multiplicity for a component reference MUST only be equal or further restrict any value for the multiplicity of the reference with the same name in the componentType of the implementation, where further restriction means 0..n to 0..1 or 1..n to 1..1.
- [ASM50010] If @wiredByImpl="true" is set for a reference, then the reference MUST NOT be wired statically within a composite, but left unwired.
- [ASM50011] If an interface is declared for a component reference it MUST provide a compatible superset of the interface declared for the equivalent reference in the componentType of the implementation, i.e. provide the same operations or a superset of the operations defined by the implementation for the reference.
- [ASM50012] If no binding elements are specified for the reference, then the bindings specified for the equivalent reference in the componentType of the implementation MUST be used, but if the componentType also has no bindings specified, then <binding.sca/> MUST be used as the binding. If binding elements are specified for the reference, then those bindings MUST be used and they override any bindings specified for the equivalent reference in the componentType of the implementation.
- [ASM50013] If @wiredByImpl="true", other methods of specifying the target service MUST NOT be used.
- [ASM50014] If @autowire="true", the autowire procedure MUST only be used if no target is identified by any of the other ways listed above. It is not an error if @autowire="true" and a target is also defined through some other means, however in this case the autowire procedure MUST NOT be used.
- [ASM50015] If a binding element has a value specified for a target service using its @uri attribute, the binding element MUST NOT identify target services using binding specific attributes or elements.
- [ASM50016] It is possible that a particular binding type MAY require that the address of a target service uses more than a simple URI. In such cases, the @uri attribute MUST NOT be used to identify the target service - instead, binding specific attributes and/or child elements must be used.
- [ASM50018] A reference with multiplicity 0..1 or 0..n MAY have no target service defined.
- [ASM50019] A reference with multiplicity 0..1 or 1..1 MUST NOT have more than one target service defined.
- [ASM50020] A reference with multiplicity 1..1 or 1..n MUST have at least one target service defined.
- [ASM50021] A reference with multiplicity 0..n or 1..n MAY have one or more target services defined.
- [ASM50022] Where it is detected that the rules for the number of target services for a reference have been violated, either at deployment or at execution time, an SCA Runtime MUST generate an error no later than when the reference is invoked by the component implementation.
- [ASM50023] Some reference multiplicity errors can be detected at deployment time. In these cases, an error SHOULD be generated by the SCA runtime at deployment time.
- [ASM50024] Other reference multiplicity errors can only be checked at runtime. In these cases, the SCA runtime MUST generate an error no later than when the reference is invoked by the component implementation.
- [ASM50025] Where a component reference is promoted by a composite reference, the promotion MUST be treated from a multiplicity perspective as providing 0 or more target services for the component reference, depending upon the further configuration of the composite reference. These target services are in addition to any target services identified on the component reference itself, subject to the rules relating to multiplicity.
- [ASM50026] If a reference has a value specified for one or more target services in its @target attribute, there MUST NOT be any child <binding/> elements declared for that reference.

- [ASM50027] If the @value attribute of a component property element is declared, the type of the property MUST be an XML Schema simple type and the @value attribute MUST contain a single value of that type.
- [ASM50028] If the value subelement of a component property is specified, the type of the property MUST be an XML Schema simple type or an XML schema complex type.
- [ASM50029] If a component property value is declared using a child element of the <property/> element, the type of the property MUST be an XML Schema global element and the declared child element MUST be an instance of that global element.
- [ASM50030] A <component/> element MUST NOT contain two <property/> subelements with the same value of the @name attribute.
- [ASM50031] The name attribute of a component property MUST match the name of a property element in the component type of the component implementation.
- [ASM50032] If a property is single-valued, the <value/> subelement MUST NOT occur more than once.
- [ASM50033] A property <value/> subelement MUST NOT be used when the @value attribute is used to specify the value for that property.
- [ASM50034] If any <wire/> element with its @replace attribute set to "true" has a particular reference specified in its @source attribute, the value of the @target attribute for that reference MUST be ignored and MUST NOT be used to define target services for that reference.
- [ASM60001] A composite name must be unique within the namespace of the composite.
- [ASM60002] @local="true" for a composite means that all the components within the composite MUST run in the same operating system process.
- [ASM60003] The name of a composite <service/> element MUST be unique across all the composite services in the composite.
- [ASM60004] A composite <service/> element's promote attribute MUST identify one of the component services within that composite.
- [ASM60005] If a composite service **interface** is specified it must be the same or a compatible subset of the interface provided by the promoted component service, i.e. provide a subset of the operations defined by the component service.
- [ASM60006] The name of a composite <reference/> element MUST be unique across all the composite references in the composite.
- [ASM60007] Each of the URIs declared by a composite reference's @promote attribute MUST identify a component reference within the composite.
- [ASM60008] the interfaces of the component references promoted by a composite reference MUST be the same, or if the composite reference itself declares an interface then all the component reference interfaces must be compatible with the composite reference interface. Compatible means that the component reference interface is the same or is a strict subset of the composite reference interface.
- [ASM60009] the intents declared on a composite reference and on the component references which it promotes MUST NOT be mutually exclusive.
- [ASM60010] If any intents in the set which apply to a composite reference are mutually exclusive then the SCA runtime MUST raise an error.
- [ASM60011] The value specified for the **multiplicity** attribute of a composite reference MUST be compatible with the multiplicity specified on each of the promoted component references, i.e. the multiplicity has to be equal or further restrict. So multiplicity 0..1 can be used where the promoted component reference has multiplicity 0..n, multiplicity 1..1 can be used where the promoted component reference has multiplicity 0..n or 1..n and multiplicity 1..n can be used where the promoted component reference has multiplicity 0..n., However, a composite reference of multiplicity 0..n or 1..n cannot be used to promote a component reference of multiplicity 0..1 or 1..1 respectively.
- [ASM60012] If a composite reference has an **interface** specified, it MUST provide an interface which is the same or which is a compatible superset of the interface(s) declared

- by the promoted component reference(s), i.e. provide a superset of the operations in the interface defined by the component for the reference.
- [ASM60013] If no interface is declared on a composite reference, the interface from one of its promoted component references is used, which MUST be the same as or a compatible superset of the interface(s) declared by the promoted component reference(s).
- [ASM60014] The name attribute of a composite property MUST be unique amongst the properties of the same composite.
- [ASM60015] the source interface and the target interface of a wire MUST either both be remotable or else both be local
- [ASM60016] the operations on the target interface of a wire MUST be the same as or be a superset of the operations in the interface specified on the source
- [ASM60017] compatibility between the source interface and the target interface for a wire for the individual operations is defined as compatibility of the signature, that is operation name, input types, and output types MUST be the same.
- [ASM60018] the order of the input and output types for operations in the source interface and the target interface of a wire also MUST be the same.
- [ASM60019] the set of Faults and Exceptions expected by each operation in the source interface MUST be the same or be a superset of those specified by the target interface.
- [ASM60020] other specified attributes of the source interface and the target interface of a wire MUST match, including Scope and Callback interface
- [ASM60021] For the case of an un-wired reference with multiplicity 1..1 or 1..n the deployment process provided by an SCA runtime SHOULD issue a warning.
- [ASM60022] For each component reference for which autowire is enabled, the the SCA runtime MUST search within the composite for target services which are compatible with the reference.
- [ASM60023] the target service interface MUST be a compatible superset of the reference interface when using autowire to wire a reference (as defined in [the section on Wires](#))
- [ASM60024] the intents, and policies applied to the service MUST be compatible with those on the reference when using autowire to wire a reference – so that wiring the reference to the service will not cause an error due to policy mismatch
- [ASM60025] for an autowire reference with multiplicity 0..1 or 1..1, the SCA runtime MUST wire the reference to one of the set of valid target services chosen from the set in a runtime-dependent fashion
- [ASM60026] for an autowire reference with multiplicity 0..n or 1..n, the reference MUST be wired to all of the set of valid target services
- [ASM60027] for an autowire reference with multiplicity 0..1 or 0..n, if the SCA runtime finds no valid target service, there is no problem – no services are wired and the SCA runtime MUST NOT raise an error
- [ASM60028] for an autowire reference with multiplicity 1..1 or 1..n, if the SCA runtime finds no valid target services an error MUST be raised by the SCA runtime since the reference is intended to be wired
- [ASM60030] The @name attribute of an <implementation.composite/> element MUST contain the QName of a composite in the SCA Domain.
- [ASM60031] The SCA runtime MUST raise an error if the composite resulting from the inclusion of one composite into another is invalid.
- [ASM60032] For a composite used as a component implementation, each composite service offered by the composite MUST promote a component service of a component that is within the composite.
- [ASM60033] For a composite used as a component implementation, every component reference of components within the composite with a multiplicity of 1..1 or 1..n MUST be wired or promoted

(according to the various rules for specifying target services for a component reference described in section 5.3.1).

- [ASM60034] For a composite used as a component implementation, all properties of components within the composite, where the underlying component implementation specifies "mustSupply=true" for the property, MUST either specify a value for the property or source the value from a composite property.
- [ASM70001] The constrainingType specifies the services, references and properties that MUST be implemented by the implementation of the component to which the constrainingType is attached.
- [ASM70002] If the configuration of the component or its implementation do not conform to the constrainingType specified on the component element, the SCA runtime MUST raise an error.
- [ASM70003] The name attribute of the constraining type MUST be unique in the SCA domain.
- [ASM70004] When an implementation is constrained by a constrainingType its component type MUST contain all the services, references and properties specified in the constrainingType.
- [ASM70005] An implementation MAY contain additional services, additional optional references (multiplicity 0..1 or 0..n) and additional optional properties beyond those declared in the constraining type, but MUST NOT contain additional non-optional references (multiplicity 1..1 or 1..n) or additional non-optional properties (a property with mustSupply=true).
- [ASM70006] Additional services, references and properties provided by the implementation which are not declared in the constrainingType associated with a component MUST NOT be configured in any way by the containing composite.
- [ASM70007] A component or implementation can use a qualified form of an intent specified in unqualified form in the constrainingType, but if the constrainingType uses the qualified form of an intent, then the component or implementation MUST also use the qualified form, otherwise there is an error.
- [ASM80001] The interface.wsdl @interface attribute MUST reference a portType of a WSDL 1.1 document.
- [ASM80002] Remotable service Interfaces MUST NOT make use of **method or operation overloading**.
- [ASM80003] If a remotable service is called locally or remotely, the SCA container MUST ensure sure that no modification of input messages by the service or post-invocation modifications to return messages are seen by the caller.
- [ASM80004] If a reference is defined using a bidirectional interface element, the client component implementation using the reference calls the referenced service using the interface. The client MUST provide an implementation of the callback interface.
- [ASM80005] Either both interfaces of a bidirectional service MUST be remotable, or both MUST be local. A bidirectional service MUST NOT mix local and remote services.
- [ASM80006] Where a service or a reference has a conversational interface, the conversational intent MUST be attached either to the interface itself, or to the service or reference using the interface.
- [ASM80007] Once an operation marked with endsConversation has been invoked, any subsequent attempts to call an operation or a callback operation associated with the same conversation MUST generate a sca:ConversationViolation fault.
- [ASM80008] Any service or reference that uses an interface marked with required intents MUST implicitly add those intents to its own @requires list.
- [ASM80009] In a bidirectional interface, the service interface can have more than one operation defined, and the callback interface can also have more than one operation defined. SCA runtimes MUST allow an invocation of any operation on the service interface to be followed by zero, one or many invocations of any of the operations on the callback interface.
- [ASM80010] Whenever an interface document declaring a callback interface is used in the

- declaration of an <interface/> element in SCA, it MUST be treated as being bidirectional with the declared callback interface.
- [ASM80011] If an <interface/> element references an interface document which declares a callback interface and also itself contains a declaration of a callback interface, the two callback interfaces MUST be compatible.
- [ASM80012] Where a component uses an implementation and the component configuration explicitly declares an interface for a service or a reference, if the matching service or reference declaration in the component type declares an interface which has a callback interface, then the component interface declaration MUST also declare a compatible interface with a compatible callback interface.
- [ASM80013] If the service or reference declaration in the component type declares an interface without a callback interface, then the component configuration for the corresponding service or reference MUST NOT declare an interface with a callback interface.
- [ASM80014] Where a composite declares an interface for a composite service or a composite reference, if the promoted service or promoted reference has an interface which has a callback interface, then the interface declaration for the composite service or the composite reference MUST also declare a compatible interface with a compatible callback interface.
- [ASM80015] If the promoted service or promoted reference has an interface without a callback interface, then the interface declaration for the composite service or composite reference MUST NOT declare a callback interface.
- [ASM80016] The interface.wsdl @callbackInterface attribute, if present, MUST reference a portType of a WSDL 1.1 document.
- [ASM90001] For a binding of a **reference** the URI attribute defines the target URI of the reference. This MUST be either the componentName/serviceName for a wire to an endpoint within the SCA domain, or the accessible address of some service endpoint either inside or outside the SCA domain (where the addressing scheme is defined by the type of the binding).
- [ASM90002] When a service or reference has multiple bindings, only one binding can have the default name value; all others must have a name value specified that is unique within the service or reference.
- [ASM90003] If a reference has any bindings they MUST be resolved which means that each binding MUST include a value for the @URI attribute or MUST otherwise specify an endpoint. The reference MUST NOT be wired using other SCA mechanisms.
- [ASM90004] a wire target MAY be specified with a syntax of "componentName/serviceName/bindingName".
- [ASM10001] all of the QNames for the definitions contained in definitions.xml files MUST be unique within the domain.
- [ASM12001] For any contribution packaging it MUST be possible to present the artifacts of the packaging to SCA as a hierarchy of resources based off of a single root
- [ASM12002] Within any contribution packaging A directory resource SHOULD exist at the root of the hierarchy named META-INF
- [ASM12003] Within any contribution packaging 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.
- [ASM12004] Optionally, in the sca-contribution.xml file, additional elements MAY exist that list the namespaces of constructs that are needed by the contribution and which are be found elsewhere, for example in other contributions.
- [ASM12005] Where present, these mechanisms MUST be used by the SCA runtime to resolve artifact dependencies.
- [ASM12006] SCA requires that all runtimes MUST support the ZIP packaging format for contributions.
- [ASM12007] Implementations of SCA MAY also generate an error if there are conflicting names exported from multiple contributions.

- [ASM12008] SCA runtimes MAY choose not to provide the contribution functions functionality in any way.
- [ASM12009] if there is ever a conflict between two indirect dependent contributions, then the conflict MUST be resolved by an explicit entry in the dependent contribution list.
- [ASM12010] Where present, non-SCA artifact resolution mechanisms MUST be used by the SCA runtime in precedence to the SCA mechanisms.
- [ASM12011] If one of the non-SCA artifact resolution mechanisms is present, but there is a failure to find the resource indicated when using the mechanism (eg the URI is incorrect or invalid, say) the SCA runtime MUST raise an error and MUST NOT attempt to use SCA resolution mechanisms as an alternative.
- [ASM12012] The value of @autowire for the logical domain composite MUST be autowire="false".
- [ASM12013] For components at the Domain level, with References for which @autowire="true" applies, the behaviour of the SCA runtime for a given Domain MUST take ONE of the 3 following forms:
- 1) The SCA runtime MAY disallow deployment of any components with autowire References. In this case, the SCA runtime MUST generate an exception at the point where the component is deployed.
  - 2) The SCA runtime MAY evaluate the target(s) for the reference at the time that the component is deployed and not update those targets when later deployment actions occur.
  - 3) The SCA runtime MAY re-evaluate the target(s) for the reference dynamically as later deployment actions occur resulting in updated reference targets which match the new Domain configuration. How the new configuration of the reference takes place is described by the relevant client and implementation specifications.
- [ASM12014] Where <wire/> elements are added, removed or replaced by deployment actions, the components whose references are affected by those deployment actions MAY have their references updated by the SCA runtime dynamically without the need to stop and start those components.
- [ASM12015] Where components are updated by deployment actions (their configuration is changed in some way, which may include changing the wires of component references), the new configuration MUST apply to all new instances of those components once the update is complete.
- [ASM12016] An SCA runtime MAY choose to maintain existing instances with the old configuration of components updated by deployment actions, but an SCA runtime MAY choose to stop and discard existing instances of those components.
- [ASM12017] Where a component that is the target of a wire is removed, without the wire being changed, then future invocations of the reference that use that wire SHOULD fail with a ServiceUnavailable fault. If the wire is the result of the autowire process, the SCA runtime MUST:
- either cause future invocation of the target component's services to fail with a ServiceUnavailable fault
  - or alternatively, if an alternative target component is available that satisfies the autowire process, update the reference of the source component
- [ASM12018] Where a component that is the target of a wire is updated, future invocations of that reference SHOULD use the updated component.
- [ASM12019] Where an existing domain level component is updated, an SCA runtime MAY maintain a copy of a component offering a conversational service until all existing conversations complete - alternatively all existing conversations MAY be terminated.
- [ASM12020] Where a component is added to the domain that is a potential target for a domain level component reference where that reference is marked as @autowire=true, the SCA runtime MUST:
- either update the references for the source component once the new component is running.
- or alternatively, defer the updating of the references of the source component until

the source component is stopped and restarted.

[ASM12021]

The SCA runtime MUST raise an error if an artifact cannot be resolved using these mechanisms, if present.

[ASM12022]

There can be multiple import declarations for a given namespace. Where multiple import declarations are made for the same namespace, all the locations specified MUST be searched in lexical order.

[ASM12023]

When a contribution contains a reference to an artifact from a namespace that is declared in an import statement of the contribution, if the SCA artifact resolution mechanism is used to resolve the artifact, the SCA runtime MUST resolve artifacts in the following order:

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

[ASM12024]

The SCA runtime MUST ignore local definitions of an artifact if the artifact is found through resolving an import statement.

[ASM12025]

The SCA runtime MUST raise an error if an artifact cannot be resolved by the precedence order above.

---

4982 **D. Acknowledgements**

4983 The following individuals have participated in the creation of this specification and are gratefully  
4984 acknowledged:

4985 **Participants:**

4986 [Participant Name, Affiliation | Individual Member]

4987 [Participant Name, Affiliation | Individual Member]

4988

---

## **E. Non-Normative Text**

4990

## F. Revision History

4991 [optional; should not be included in OASIS Standards]

4992

Revision	Date	Editor	Changes Made
1	2007-09-24	Anish Karmarkar	Applied the OASIS template + related changes to the Submission
2	2008-01-04	Michael Beisiegel	<p>composite section</p> <ul style="list-style-type: none"> <li>- changed order of subsections from property, reference, service to service, reference, property</li> <li>- progressive disclosure of pseudo schemas, each section only shows what is described</li> <li>- attributes description now starts with name : type (cardinality)</li> <li>- child element description as list, each item starting with name : type (cardinality)</li> <li>- added section in appendix to contain complete pseudo schema of composite</li> </ul> <p>implementation section</p> <ul style="list-style-type: none"> <li>- moved component section after implementation section</li> <li>- made the ConstrainingType section a top level section</li> <li>- moved interface section to after constraining type section</li> </ul> <p>component section</p> <ul style="list-style-type: none"> <li>- added subheadings for Implementation, Service, Reference, Property</li> <li>- progressive disclosure of pseudo schemas, each section only shows what is described</li> <li>- attributes description now starts with name : type (cardinality)</li> <li>- child element description as list, each item starting with name : type (cardinality)</li> </ul> <p>implementation section</p> <ul style="list-style-type: none"> <li>- changed title to "Implementation and ComponentType"</li> <li>- moved implementation instance related stuff from implementation section to component implementation section</li> <li>- added subheadings for Service, Reference, Property, Implementation</li> <li>- progressive disclosure of pseudo schemas, each section only shows what is described</li> <li>- attributes description now starts with name : type (cardinality)</li> <li>- child element description as list, each item starting with name : type (cardinality)</li> <li>- attribute and element description still needs to be completed, all implementation statements</li> </ul>

			<p>on services, references, and properties should go here</p> <ul style="list-style-type: none"> <li>- added complete pseudo schema of componentType in appendix</li> <li>- added "Quick Tour by Sample" section, no content yet</li> <li>- added comment to introduction section that the following text needs to be added <ul style="list-style-type: none"> <li>"This specification is defined in terms of infoset and not XML 1.0, even though the spec uses XML 1.0/1.1 terminology. A mapping from XML to infoset (... link to infoset specification ...) is trivial and should be used for non-XML serializations."</li> </ul> </li> </ul>
3	2008-02-15	Anish Karmarkar Michael Beisiegel	<p>Incorporated resolutions from 2008 Jan f2f.</p> <ul style="list-style-type: none"> <li>- issue 9</li> <li>- issue 19</li> <li>- issue 21</li> <li>- issue 4</li> <li>- issue 1A</li> <li>- issue 27</li> <li>- in Implementation and ComponentType section added attribute and element description for service, reference, and property</li> <li>- removed comments that helped understand the initial restructuring for WD02</li> <li>- added changes for issue 43</li> <li>- added changes for issue 45, except the changes for policySet and requires attribute on property elements</li> <li>- used the NS <a href="http://docs.oasis-open.org/ns/opencsa/sca/200712">http://docs.oasis-open.org/ns/opencsa/sca/200712</a></li> <li>- updated copyright stmt</li> <li>- added wordings to make PDF normative and xml schema at the NS uri authoritative</li> </ul>
4	2008-04-22	Mike Edwards	<p>Editorial tweaks for CD01 publication:</p> <ul style="list-style-type: none"> <li>- updated URL for spec documents</li> <li>- removed comments from published CD01 version</li> <li>- removed blank pages from body of spec</li> </ul>
5	2008-06-30	Anish Karmarkar Michael Beisiegel	<p>Incorporated resolutions of issues: 3, 6, 14 (only as it applies to the component property element), 23, 25, 28, 25, 38, 39, 40, 42, 45 (except for adding @requires and @policySets to property elements), 57, 67, 68, 69</p>
6	2008-09-23	Mike Edwards	<p>Editorial fixes in response to Mark Combellack's review contained in email: <a href="http://lists.oasis-open.org/archives/sca-assembly/200804/msg00089.html">http://lists.oasis-open.org/archives/sca-assembly/200804/msg00089.html</a></p>
7 CD01 - Rev3	2008-11-18	Mike Edwards	<ul style="list-style-type: none"> <li>• Specification marked for conformance statements. New Appendix (D) added</li> </ul>

			containing a table of all conformance statements. Mass of related minor editorial changes to remove the use of RFC2119 words where not appropriate.
8 CD01 - Rev4	2008-12-11	Mike Edwards	<ul style="list-style-type: none"> <li>- Fix problems of misplaced statements in Appendix D</li> <li>- Fixed problems in the application of Issue 57 - section 5.3.1 &amp; Appendix D as defined in email: <a href="http://lists.oasis-open.org/archives/sca-assembly/200811/msg00045.html">http://lists.oasis-open.org/archives/sca-assembly/200811/msg00045.html</a></li> <li>- Added Conventions section, 1.3, as required by resolution of Issue 96.</li> <li>- Issue 32 applied - section B2</li> <li>- Editorial addition to section 8.1 relating to no operation overloading for remotable interfaces, as agreed at TC meeting of 16/09/2008.</li> </ul>
9 CD01 - Rev5	2008-12-22	Mike Edwards	<ul style="list-style-type: none"> <li>- Schemas in Appendix B updated with resolutions of Issues 32 and 60</li> <li>- Schema for contributions - Appendix B12 - updated with resolutions of Issues 53 and 74.</li> <li>- Issues 53 and 74 incorporated - Sections 11.4, 11.5</li> </ul>
10 CD01-Rev6	2008-12-23	Mike Edwards	<ul style="list-style-type: none"> <li>- Issues 5, 71, 92</li> <li>- Issue 14 - remaining updates applied to ComponentType (section 4.1.3) and to Composite Property (section 6.3)</li> </ul>
11 CD01-Rev7	2008-12-23	Mike Edwards	<p>All changes accepted before revision from Rev6 started - due to changes being applied to previously changed sections in the Schemas</p> <ul style="list-style-type: none"> <li>Issues 12 &amp; 18 - Section B2</li> <li>Issue 63 - Section C3</li> <li>Issue 75 - Section C12</li> <li>Issue 65 - Section 7.0</li> <li>Issue 77 - Section 8 + Appendix D</li> <li>Issue 69 - Sections 5.1, 8</li> <li>Issue 45 - Sections 4.1.3, 5.4, 6.3, B2.</li> <li>Issue 56 - Section 8.2, Appendix D</li> <li>Issue 41 - Sections 5.3.1, 6.4, 12.7, 12.8, Appendix D</li> </ul>
12 CD01-Rev8	2008-12-30	Mike Edwards	<ul style="list-style-type: none"> <li>Issue 72 - Removed Appendix A</li> <li>Issue 79 - Sections 9.0, 9.2, 9.3, Appendix A.2</li> <li>Issue 62 - Sections 4.1.3, 5.4</li> <li>Issue 26 - Section 6.5</li> <li>Issue 51 - Section 6.5</li> <li>Issue 36 - Section 4.1</li> <li>Issue 44 - Section 10, Appendix C</li> <li>Issue 89 - Section 8.2, 8.5, Appendix A, Appendix C</li> <li>Issue 16 - Section 6.8, 9.4</li> <li>Issue 8 - Section 11.2.1</li> <li>Issue 17 - Section 6.6</li> <li>Issue 30 - Sections 4.1.1, 4.1.2, 5.2, 5.3, 6.1, 6.2, 9</li> <li>Issue 33 - insert new Section 8.4</li> </ul>
12 CD01-Rev8a	2009-01-13	Bryan Aupperle Mike Edwards	Issue 99 - Section 8

13 CD02	2009-01-14	Mike Edwards	All changes accepted All comments removed
---------	------------	--------------	----------------------------------------------

4993