

SCA Service Component Architecture

Assembly Model Specification

SCA Version 1.00, March 15 2007

| | | |
|---------------------|-------------------|------------------------|
| Technical Contacts: | Michael Beisiegel | IBM Corporation |
| | Henning Blohm | SAP AG |
| | Dave Booz | IBM Corporation |
| | Mike Edwards | IBM Corporation |
| | Oisin Hurley | IONA Technologies plc. |
| | Sabin Ielceanu | TIBCO Software Inc. |
| | Alex Miller | BEA Systems, Inc. |
| | Anish Karmarkar | Oracle |
| | Ashok Malhotra | Oracle |
| | Jim Marino | BEA Systems, Inc. |
| | Martin Nally | IBM Corporation |
| | Eric Newcomer | IONA Technologies plc. |
| | Sanjay Patil | SAP AG |
| | Greg Pavlik | Oracle |
| | Martin Raeppe | SAP AG |
| | Michael Rowley | BEA Systems, Inc. |
| | Ken Tam | BEA Systems, Inc. |
| | Scott Vorthmann | TIBCO Software Inc. |
| | Peter Walker | Sun Microsystems Inc. |
| | Lance Waterman | Sybase, Inc. |

Copyright Notice

© Copyright BEA Systems, Inc., Cape Clear Software, International Business Machines Corp, Interface21, IONA Technologies, Oracle, Primeton Technologies, Progress Software, Red Hat, Rogue Wave Software, SAP AG., Siemens AG., Software AG., Sun Microsystems, Inc., Sybase Inc., TIBCO Software Inc., 2005, 2007. All rights reserved.

License

The Service Component Architecture Specification is being provided by the copyright holders under the following license. By using and/or copying this work, you agree that you have read, understood and will comply with the following terms and conditions:

Permission to copy, display and distribute the Service Component Architecture Specification and/or portions thereof, without modification, in any medium without fee or royalty is hereby granted, provided that you include the following on ALL copies of the Service Component Architecture Specification, or portions thereof, that you make:

1. A link or URL to the Service Component Architecture Specification at this location:
 - <http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications>
2. The full text of the copyright notice as shown in the Service Component Architecture Specification.

BEA, Cape Clear, IBM, Interface21, IONA, Oracle, Primeton, Progress Software, Red Hat, Rogue Wave, SAP, Siemens, Software AG., Sun, Sybase, TIBCO (collectively, the "Authors") agree to grant you a royalty-free license, under reasonable, non-discriminatory terms and conditions to patents that they deem necessary to implement the Service Component Architecture Specification.

THE Service Component Architecture SPECIFICATION IS PROVIDED "AS IS," AND THE AUTHORS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, REGARDING THIS SPECIFICATION AND THE IMPLEMENTATION OF ITS CONTENTS, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT OR TITLE.

THE AUTHORS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATING TO ANY USE OR DISTRIBUTION OF THE Service Components Architecture SPECIFICATION.

The name and trademarks of the Authors may NOT be used in any manner, including advertising or publicity pertaining to the Service Component Architecture Specification or its contents without specific, written prior permission. Title to copyright in the Service Component Architecture Specification will at all times remain with the Authors.

No other rights are granted by implication, estoppel or otherwise.

Status of this Document

This specification may change before final release and you are cautioned against relying on the content of this specification. The authors are currently soliciting your contributions and suggestions. Licenses are available for the purposes of feedback and (optionally) for implementation.

IBM is a registered trademark of International Business Machines Corporation in the United States, other countries, or both.

BEA is a registered trademark of BEA Systems, Inc.

Cape Clear is a registered trademark of Cape Clear Software

IONA and IONA Technologies are registered trademarks of IONA Technologies plc.

Oracle is a registered trademark of Oracle USA, Inc.

Progress is a registered trademark of Progress Software Corporation

Primeton is a registered trademark of Primeton Technologies, Ltd.

Red Hat is a registered trademark of Red Hat Inc.

Rogue Wave is a registered trademark of Quovadx, Inc

SAP is a registered trademark of SAP AG.

SIEMENS is a registered trademark of SIEMENS AG

Software AG is a registered trademark of Software AG

Sun and Sun Microsystems are registered trademarks of Sun Microsystems, Inc.

Sybase is a registered trademark of Sybase, Inc.

TIBCO is a registered trademark of TIBCO Software Inc.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Table of Contents

| | |
|----------------------------------------------------------------------------------|-----|
| SCA Service Component Architecture..... | i |
| Copyright Notice | ii |
| License | ii |
| Status of this Document..... | iii |
| 1 Assembly Model | 1 |
| 1.1 Introduction | 1 |
| 1.2 Overview | 1 |
| 1.2.1 Diagrams used to Represent SCA Artifacts | 3 |
| 1.3 Component | 5 |
| 1.3.1 Example Component | 9 |
| 1.4 Implementation | 11 |
| 1.4.1 Component Type | 13 |
| 1.4.1.1 Example ComponentType | 15 |
| 1.4.1.2 Example Implementation | 15 |
| 1.5 Interface | 17 |
| 1.5.1 Local and Remotable Interfaces | 18 |
| 1.5.2 Bidirectional Interfaces..... | 19 |
| 1.5.3 Conversational Interfaces | 20 |
| 1.5.4 SCA-Specific Aspects for WSDL Interfaces..... | 22 |
| 1.6 Composite | 23 |
| 1.6.1 Property – Definition and Configuration | 26 |
| 1.6.1.1 Property Examples | 27 |
| 1.6.2 References | 30 |
| 1.6.2.1 Example Reference | 32 |
| 1.6.3 Service | 34 |
| 1.6.3.1 Service Examples..... | 35 |
| 1.6.4 Wire | 36 |
| 1.6.4.1 Wire Examples..... | 38 |
| 1.6.4.2 Autowire..... | 40 |
| 1.6.4.3 Autowire Examples..... | 41 |
| 1.6.5 Using Composites as Component Implementations | 43 |
| 1.6.5.1 Example of Composite used as a Component Implementation..... | 44 |
| 1.6.6 Using Composites through Inclusion | 45 |
| 1.6.6.1 Included Composite Examples..... | 46 |
| 1.6.7 Composites which Include Component Implementations of Multiple Types | 48 |

| | | |
|----------|---------------------------------------------------------------------|----|
| 1.6.8 | ConstrainingType | 48 |
| 1.6.8.1 | Example constrainingType | 50 |
| 1.7 | Binding..... | 51 |
| 1.7.1 | Messages containing Data not defined in the Service Interface | 52 |
| 1.7.2 | Form of the URI of a Deployed Binding | 53 |
| 1.7.2.1 | Constructing Hierarchical URIs | 53 |
| 1.7.2.2 | Non-hierarchical URIs | 54 |
| 1.7.2.3 | Determining the URI scheme of a deployed binding..... | 54 |
| 1.7.3 | SCA Binding | 54 |
| 1.7.3.1 | Example SCA Binding | 55 |
| 1.7.4 | Web Service Binding | 55 |
| 1.7.5 | JMS Binding..... | 56 |
| 1.8 | SCA Definitions..... | 57 |
| 1.9 | Extension Model | 58 |
| 1.9.1 | Defining an Interface Type..... | 58 |
| 1.9.2 | Defining an Implementation Type | 59 |
| 1.9.3 | Defining a Binding Type..... | 61 |
| 1.10 | Packaging and Deployment..... | 63 |
| 1.10.1 | Domains..... | 63 |
| 1.10.2 | Contributions | 63 |
| 1.10.2.1 | SCA Artifact Resolution..... | 64 |
| 1.10.2.2 | SCA Contribution Metadata Document | 65 |
| 1.10.2.3 | Contribution Packaging using ZIP | 66 |
| 1.10.3 | Installed Contribution | 67 |
| 1.10.3.1 | Installed Artifact URIs | 67 |
| 1.10.4 | Operations for Contributions..... | 67 |
| 1.10.4.1 | install Contribution & update Contribution | 68 |
| 1.10.4.2 | add Deployment Composite & update Deployment Composite | 68 |
| 1.10.4.3 | remove Contribution | 68 |
| 1.10.5 | Use of Existing (non-SCA) Mechanisms for Resolving Artifacts | 68 |
| 1.10.6 | Domain-Level Composite | 69 |
| 1.10.6.1 | add To Domain-Level Composite | 69 |
| 1.10.6.2 | remove From Domain-Level Composite | 69 |
| 1.10.6.3 | get Domain-Level Composite..... | 69 |
| 1.10.6.4 | get QName Definition | 69 |
| 2 | Appendix 1..... | 71 |
| 2.1 | XML Schemas..... | 71 |

| | | |
|---------|----------------------------------------|----|
| 2.1.1 | sca.xsd | 71 |
| 2.1.2 | sca-core.xsd | 71 |
| 2.1.3 | sca-binding-sca.xsd | 77 |
| 2.1.4 | sca-interface-java.xsd | 77 |
| 2.1.5 | sca-interface-wsdl.xsd | 78 |
| 2.1.6 | sca-implementation-java.xsd | 78 |
| 2.1.7 | sca-implementation-composite.xsd | 79 |
| 2.1.8 | sca-definitions.xsd | 79 |
| 2.1.9 | sca-binding-webservice.xsd | 80 |
| 2.1.10 | sca-binding-jms.xsd | 80 |
| 2.1.11 | sca-policy.xsd | 80 |
| 2.2 | SCA Concepts | 81 |
| 2.2.1 | Binding | 81 |
| 2.2.2 | Component | 81 |
| 2.2.3 | Service | 81 |
| 2.2.3.1 | Remotable Service | 81 |
| 2.2.3.2 | Local Service | 81 |
| 2.2.4 | Reference | 82 |
| 2.2.5 | Implementation | 82 |
| 2.2.6 | Interface | 82 |
| 2.2.7 | Composite | 82 |
| 2.2.8 | Composite inclusion | 83 |
| 2.2.9 | Property | 83 |
| 2.2.10 | Domain | 83 |
| 2.2.11 | Wire | 83 |
| 3 | References | 84 |

1 Assembly Model

1.1 Introduction

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

- A model for the assembly of services, both tightly coupled and loosely coupled

- A model for applying infrastructure capabilities to services and to service interactions, including Security and Transactions

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

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

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

1.2 Overview

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.

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

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

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

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

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

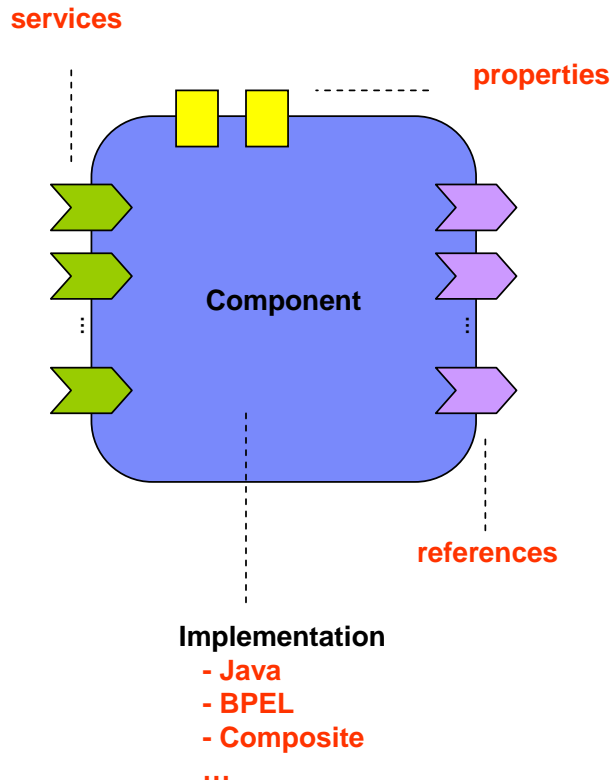
68

69 1.2.1 Diagrams used to Represent SCA Artifacts

70

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

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



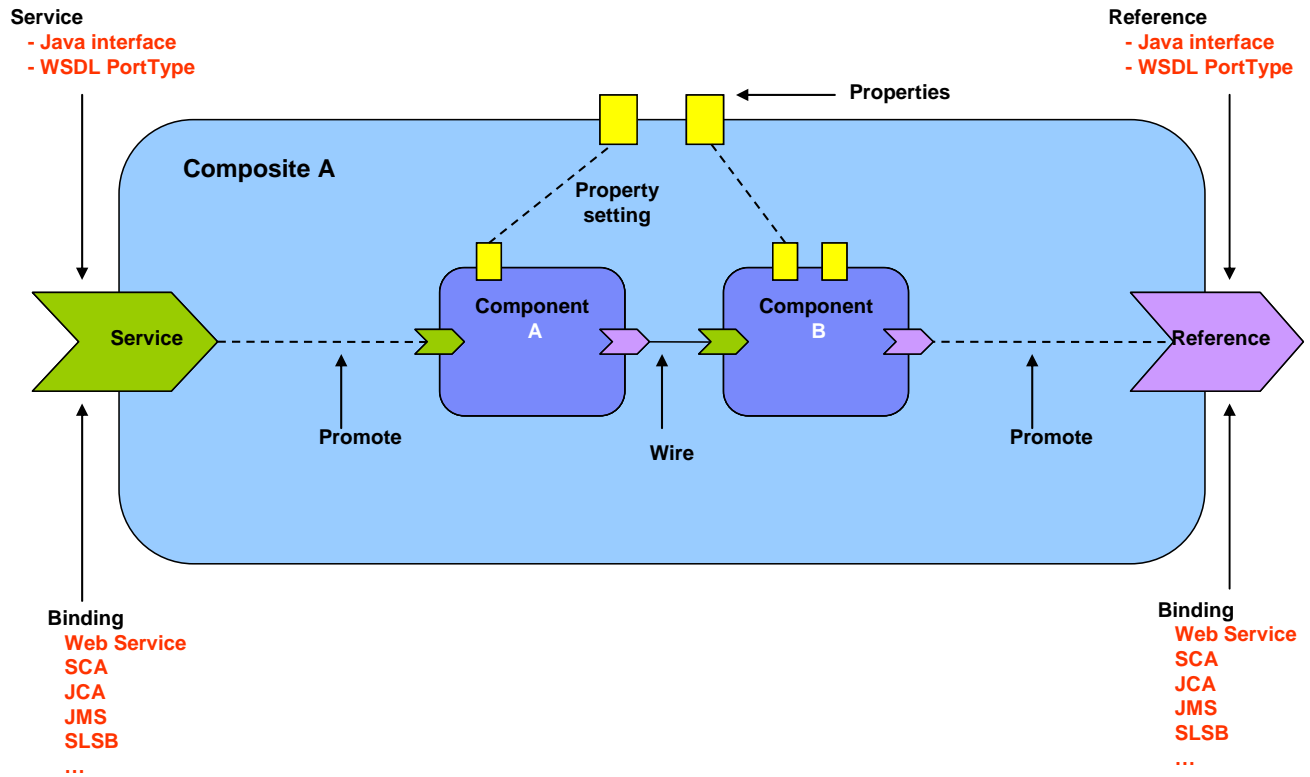
75

76 **Figure 1: SCA Component Diagram**

77

78 The following picture illustrates some of the features of a composite assembled using a set of
79 components:

80

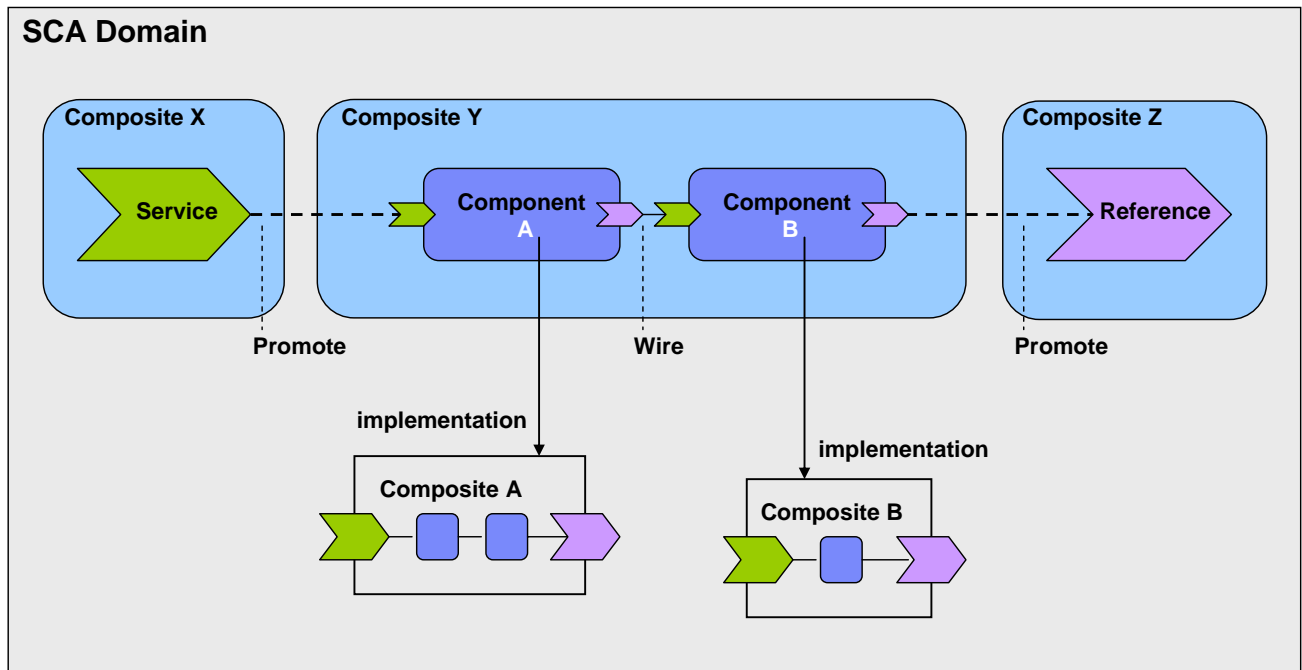


81

82 **Figure 2: SCA Composite Diagram**

83

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



86

87 **Figure 3: SCA Domain Diagram**

88

89 1.3 Component

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

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

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

99

```
100 <?xml version="1.0" encoding="UTF-8"?>
101 <!-- Component schema snippet -->
102 <composite xmlns="http://www.oesa.org/xmlns/sca/1.0"
103           targetNamespace="xs:anyURI"
104           name="xs:NCName" local="xs:boolean"?
105           autowire="xs:boolean"? constrainingType="QName"?
106           requires="list of xs:QName"? policySets="list of xs:QName"?>
107
108   ...
109
110   <component name="xs:NCName" requires="list of xs:QName"?
111             autowire="xs:boolean"?
112             requires="list of xs:QName"? policySets="list of xs:QName"?
113             constrainingType="xs:QName"?>*
114     <implementation/>?
115     <service name="xs:NCName" requires="list of xs:QName"?
116           policySets="list of xs:QName"?>*
117       <interface/>?
118       <binding uri="xs:anyURI"? requires="list of xs:QName"?
119             policySets="list of xs:QName"?/>*
120     </service>
121     <reference name="xs:NCName" multiplicity="0..1 or 1..1 or 0..n or 1..n"?
122           autowire="xs:boolean"?
123           target="list of xs:anyURI"? policySets="list of xs:QName"?
124           wiredByImpl="xs:boolean"? requires="list of xs:QName"?>*
125       <interface/>?
126       <binding uri="xs:anyURI"? requires="list of xs:QName"?
127             policySets="list of xs:QName"?/>*
128     </reference>
129     <property name="xs:NCName" (type="xs:QName" | element="xs:QName")?
130           mustSupply="xs:boolean"?
131           many="xs:boolean"? source="xs:string"? file="xs:anyURI"?>*
132           property-value?
133     </property>
134   </component>
135
136   ...
137
138 </composite>
```

141 The component element has the following *attributes*:

- 142 • **name (required)** – the name of the component. The name must be unique across all the
143 components in the composite.
- 144 • **autowire (optional)** – whether contained component references should be autowired, as
145 described in [the Autowire section](#). Default is false.
- 146 • **requires (optional)** – a list of policy intents. See the [Policy Framework specification \[10\]](#)
147 for a description of this attribute.
- 148 • **policySets (optional)** – a list of policy sets. See the [Policy Framework specification \[10\]](#)
149 for a description of this attribute.
- 150 • **constrainingType (optional)** – the name of a constrainingType. When specified, the
151 set of services, references and properties of the component, plus related intents, is
152 constrained to the set defined by the constrainingType. See [the ConstrainingType Section](#)
153 for more details.

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

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

162
163 The service element has the following **attributes**:

- 164 • **name (required)** - the name of the service. Has to match a name of a service defined
165 by the implementation.
- 166 • **requires (optional)** – a list of policy intents. See the [Policy Framework specification \[10\]](#)
167 for a description of this attribute.
168 Note: The effective set of policy intents for the service consists of any intents explicitly
169 stated in this requires attribute, combined with any intents specified for the service by the
170 implementation.
- 171 • **policySets (optional)** – a list of policy sets. See the [Policy Framework specification \[10\]](#)
172 for a description of this attribute.

173
174 A service has **zero or one interface**, which describes the operations provided by the service.
175 The interface is described by an **interface element** which is a child element of the service
176 element. If no interface is specified, then the interface specified for the service by the
177 implementation is in effect. If an interface is specified it must provide a compatible subset of the
178 interface provided by the implementation, i.e. provide a subset of the operations defined by the
179 implementation for the service. For details on the interface element see [the Interface section](#).

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

186
187 The component element can have **zero or more reference elements** as children which are
188 used to configure the references of the component. The references that can be configured are
189 defined by the implementation.

190
191 The reference element has the following *attributes*:

- 192 • ***name (required)*** – the name of the reference. Has to match a name of a reference
193 defined by the implementation.
- 194 • ***autowire (optional)*** – whether the reference should be autowired, as described in [the](#)
195 [Autowire section](#). Default is false.
- 196 • ***requires (optional)*** – a list of policy intents. See the [Policy Framework specification \[10\]](#)
197 for a description of this attribute.
198 Note: The effective set of policy intents for the reference consists of any intents explicitly
199 stated in this requires attribute, combined with any intents specified for the reference by
200 the implementation.
- 201 • ***policySets (optional)*** – a list of policy sets. See the [Policy Framework specification \[10\]](#)
202 for a description of this attribute.
- 203 • ***multiplicity (optional)*** - defines the number of wires that can connect the reference to
204 target services. Overrides the multiplicity specified for this reference on the
205 implementation. The value can only be equal or further restrict, i.e. 0..n to 0..1 or 1..n to
206 1..1. The multiplicity can have the following values
 - 207 ○ 1..1 – one wire can have the reference as a source
 - 208 ○ 0..1 – zero or one wire can have the reference as a source
 - 209 ○ 1..n – one or more wires can have the reference as a source
 - 210 ○ 0..n - zero or more wires can have the reference as a source
- 211 • ***target (optional)*** – a list of one or more of target service URI's, depending on
212 multiplicity setting. Each value wires the reference to a component service that resolves
213 the reference. For more details on wiring see [the section on Wires](#). Overrides any target
214 specified for this reference on the implementation.
- 215 • ***wiredByImpl (optional)*** – a boolean value, "false" by default, which indicates that the
216 implementation wires this reference dynamically. If set to "true" it indicates that the
217 target of the reference is set at runtime by the implementation code (eg by the code
218 obtaining an endpoint reference by some means and setting this as the target of the
219 reference through the use of programming interfaces defined by the relevant Client and
220 Implementation specification). If "true" is set, then the reference should not be wired
221 statically within a composite, but left unwired.

222
223 A reference has ***zero or one interface***, which describes the operations required by the
224 reference. The interface is described by an ***interface element*** which is a child element of the
225 reference element. If no interface is specified, then the interface specified for the reference by
226 the implementation is in effect. If an interface is specified it must provide a compatible superset
227 of the interface provided by the implementation, i.e. provide a superset of the operations defined
228 by the implementation for the reference. For details on the interface element see [the Interface](#)
229 [section](#).

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

236 Note that a binding element may specify an endpoint which is the target of that binding. A
237 reference must not mix the use of endpoints specified via binding elements with target endpoints

238 specified via the target attribute. If the target attribute is set, then binding elements can only
239 list one or more binding types that can be used for the wires identified by the target attribute.
240 All the binding types identified are available for use on each wire in this case. If endpoints are
241 specified in the binding elements, each endpoint must use the binding type of the binding
242 element in which it is defined. In addition, each binding element needs to specify an endpoint in
243 this case.

244
245 The component element has **zero or more property elements** as its children, which are used
246 to configure data values of properties of the implementation. Each property element provides a
247 value for the named property, which is passed to the implementation. The properties that can
248 be configured and their types are defined by the implementation. An implementation can declare
249 a property as multi-valued, in which case, multiple property values can be present for a given
250 property.

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

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

255
256 The form of the value of the source attribute follows the form of an XPath expression.
257 This form allows a specific property of the composite to be addressed by name. Where
258 the property is complex, the XPath expression can be extended to refer to a sub-part of
259 the complex value.

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

- 264
265 • By specifying a dereferencable URI to a file containing the property value through the **file**
266 attribute. The contents of the referenced file are used as the value of the property.

267
268 If more than one property value specification is present, the source attribute takes precedence,
269 then the file attribute.

270
271 Optionally, the type of the property can be specified in **one** of two ways:

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

274 The property type specified must be compatible with the type of the property declared by the
275 implementation. If no type is specified, the type of the property declared by the implementation
276 is used.

277
278 The property element has the following attributes:

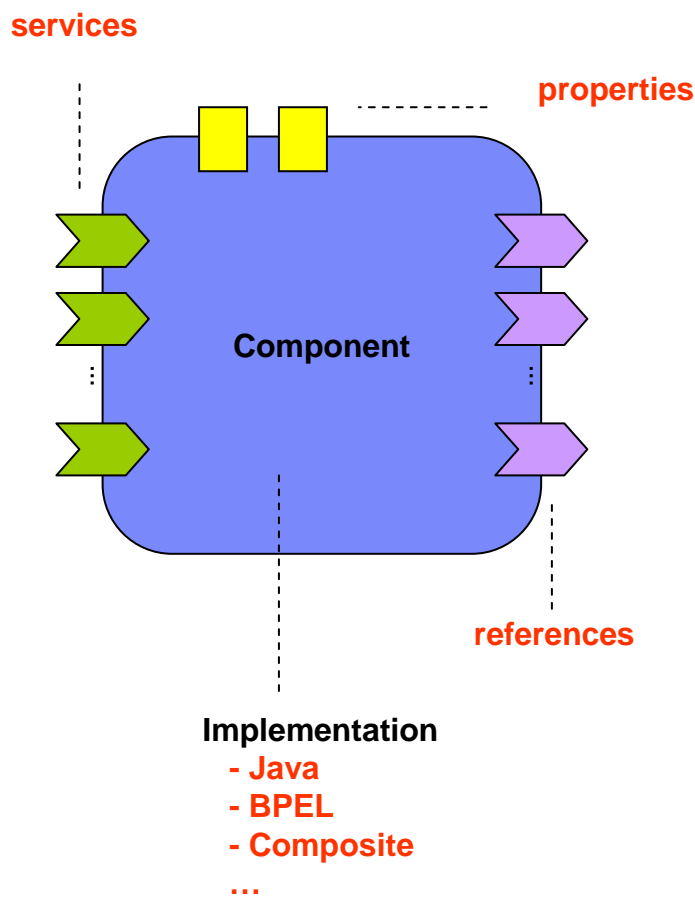
- 279 • **name (required)** – the name of the property. Has to match a name of a property
280 defined by the implementation
- 281 • **type (optional)** – the type of the property defined as the qualified name of an XML
282 schema type
- 283 • **element (optional)** – the type of the property defined as the qualified name of an XML
284 schema global element – the type is the type of the global element
- 285 • **source (optional)** – an XPath expression pointing to a property of the containing
286 composite from which the value of this component property is obtained.

287
288
289
290
291
292
293
294

- **file (optional)** – a dereferencable URI to a file containing a value for the property
- **many (optional)** – (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.

1.3.1 Example Component

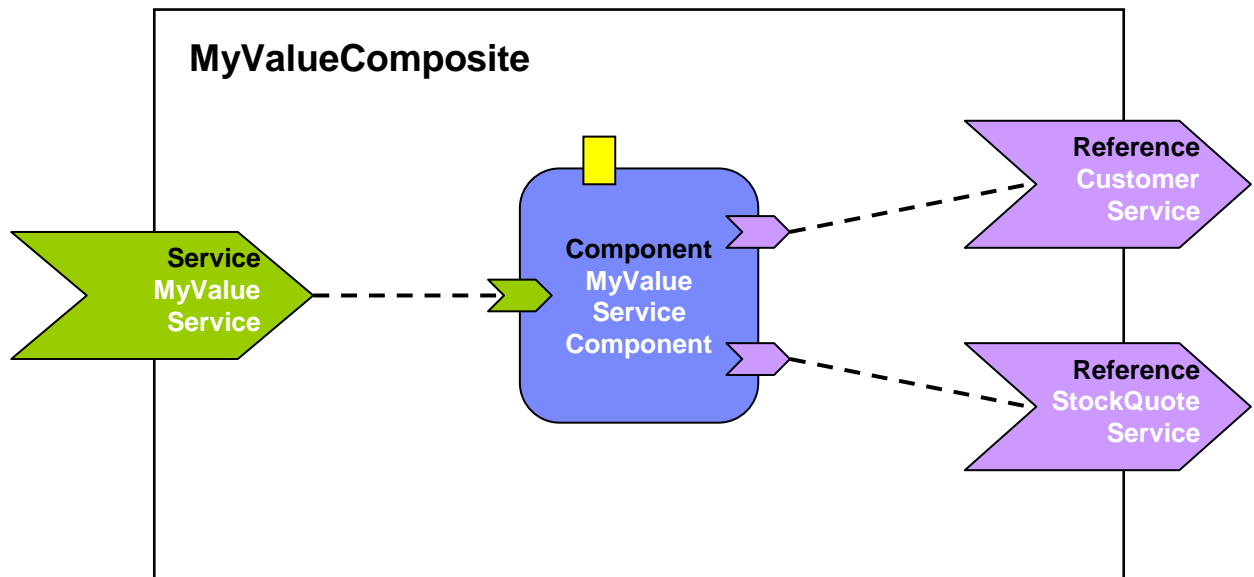
The following figure shows the **component symbol** that is used to represent a component in an assembly diagram.



299
300
301
302
303

Figure 4: Component symbol

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



304

305

306 **Figure 5: Assembly diagram for MyValueComposite**

307

308 The following snippet shows the MyValueComposite.composite file for the MyValueComposite
 309 containing the component element for the MyValueServiceComponent. A value is set for the
 310 property named currency, and the customerService and stockQuoteService references are
 311 promoted:

312

```

313 <?xml version="1.0" encoding="ASCII"?>
314 <!-- MyValueComposite_1 example -->
315 <composite xmlns="http://www.oxa.org/xmlns/sca/1.0"
316           targetNamespace="http://foo.com"
317           name="MyValueComposite" >
318
319     <service name="MyValueService" promote="MyValueServiceComponent" />
320
321     <component name="MyValueServiceComponent">
322       <implementation.java class="services.myvalue.MyValueServiceImpl" />
323       <property name="currency">EURO</property>
324       <reference name="customerService" />
325       <reference name="stockQuoteService" />
326     </component>
327
328     <reference name="CustomerService"
329               promote="MyValueServiceComponent/customerService" />
330
331     <reference name="StockQuoteService"
332               promote="MyValueServiceComponent/stockQuoteService" />
333
334 </composite>
  
```

335

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

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

```
343 <?xml version="1.0" encoding="ASCII"?>
344 <!-- MyValueComposite_2 example -->
345 <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
346           targetNamespace="http://foo.com"
347           name="MyValueComposite" >
348
349     <service name="MyValueService" promote="MyValueServiceComponent" />
350
351     <component name="MyValueServiceComponent">
352       <implementation.java class="services.myvalue.MyValueServiceImpl" />
353       <property name="currency">EURO</property>
354       <property name="currency">Yen</property>
355       <property name="currency">USDollar</property>
356       <reference name="customerService"
357                 target="InternalCustomer/customerService" />
358       <reference name="StockQuoteService" />
359     </component>
360
361     ...
362
363     <reference name="CustomerService"
364               promote="MyValueServiceComponent/customerService" />
365
366     <reference name="StockQuoteService"
367               promote="MyValueServiceComponent/StockQuoteService" />
368
369 </composite>
```

371this assumes that the composite has another component called InternalCustomer (not shown)
372 which has a service to which the customerService reference of the MyValueServiceComponent is
373 wired as well as being promoted externally through the composite reference CustomerService.

374 1.4 Implementation

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

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

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

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

```
392
393 <implementation.java class="services.myvalue.MyValueServiceImpl" />
```

```
394 <implementation.bpel process="MoneyTransferProcess"/>
395
396 <implementation.composite name="MyValueComposite"/>
397
```

398

399 **Services, references and properties** are the configurable aspects of an implementation. SCA
400 refers to them collectively as the **component type**. The characteristics of services, references
401 and properties are described in the Component section. Depending on the implementation type,
402 the implementation may be able to declare the services, references and properties that it has
403 and it also may be able to set values for all the characteristics of those services, references and
404 properties.

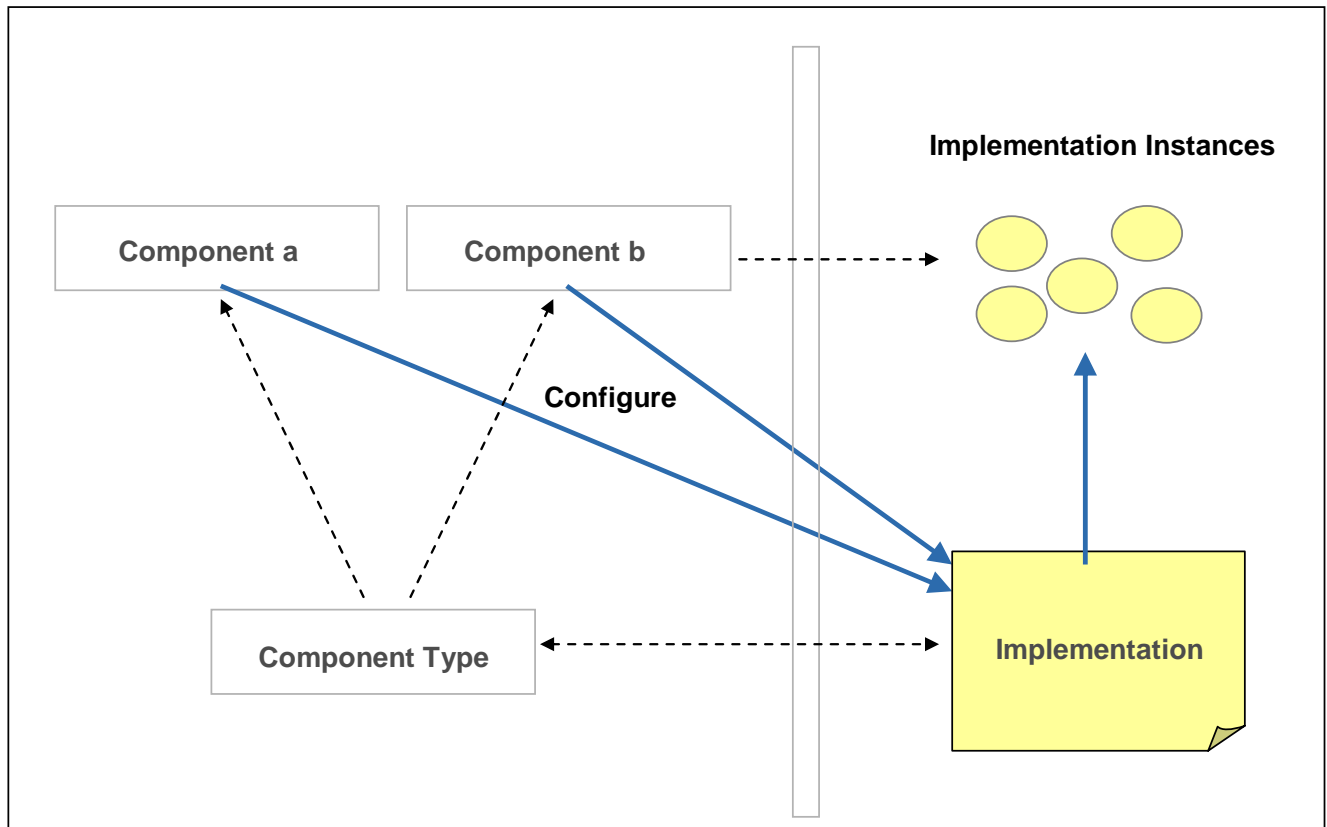
405 So, for example:

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

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

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

420 At runtime, an **implementation instance** is a specific runtime instantiation of the
421 implementation – its runtime form depends on the implementation technology used. The
422 implementation instance derives its business logic from the implementation on which it is based,
423 but the values for its properties and references are derived from the component which configures
424 the implementation.



425
426 **Figure 6: Relationship of Component and Implementation**
427

428 **1.4.1 Component Type**

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

433 The **component type is calculated in two steps** where the second step adds to the
434 information found in the first step. Step one is introspecting the implementation (if possible),
435 including the inspection of implementation annotations (if available). Step two covers the cases
436 where introspection of the implementation is not possible or where it does not provide complete
437 information and it involves looking for an SCA **component type file**. Component type
438 information found in the component type file must be compatible with the equivalent information
439 found from inspection of the implementation. The component type file can specify partial
440 information, with the remainder being derived from the implementation.

441 In the ideal case, the component type information is determined by inspecting the
442 implementation, for example as code annotations. The component type file provides a
443 mechanism for the provision of component type information for implementation types where the
444 information cannot be determined by inspecting the implementation.

445 A **component type file** has the same name as the implementation file but has the extension
446 **".componentType"**. The component type is defined by a **componentType element** in the file.
447 The **location** of the component type file depends on the type of the component implementation:
448 it is described in the respective client and implementation model specification for the
449 implementation type.

450 The componentType element can contain Service elements, Reference elements and Property
451 elements.

452 The following snippet shows the componentType schema.

453

```
454 <?xml version="1.0" encoding="ASCII"?>
455 <!-- Component type schema snippet -->
456 <componentType xmlns="http://www.oesa.org/xmlns/sca/1.0"
457   constrainingType="QName"? >
458
459   <service name="xs:NCName" requires="list of xs:QName"?
460     policySets="list of xs:QName"?>*
461     <interface/>
462     <binding uri="xs:anyURI"? requires="list of xs:QName"?
463       policySets="list of xs:QName"?/>*
464   </service>
465
466   <reference name="xs:NCName" target="list of xs:anyURI"?
467     multiplicity="0..1 or 1..1 or 0..n or 1..n"?
468     wiredByImpl="xs:boolean"? requires="list of xs:QName"?
469     policySets="list of xs:QName"?>*
470     <interface/>?
471     <binding uri="xs:anyURI"? requires="list of xs:QName"?
472       policySets="list of xs:QName"?/>*
473   </reference>
474
475   <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
476     many="xs:boolean"? mustSupply="xs:boolean"?
477     policySets="list of xs:QName"?>*
478     default-property-value?
479   </property>
480
481   <implementation requires="list of xs:QName"?
482     policySets="list of xs:QName"?/>?
483
484 </componentType>
485
```

486 ComponentType has a single attribute:

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

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

494

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

499

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

503

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

507

508 **1.4.1.1 Example ComponentType**

509

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

```
513  
514 <?xml version="1.0" encoding="ASCII"?>  
515 <componentType xmlns="http://www.osea.org/xmlns/sca/1.0">  
516  
517     <service name="MyValueService">  
518         <interface.java interface="services.myvalue.MyValueService"/>  
519     </service>  
520  
521     <reference name="customerService">  
522         <interface.java interface="services.customer.CustomerService"/>  
523     </reference>  
524     <reference name="stockQuoteService">  
525         <interface.java interface="services.stockquote.StockQuoteService"/>  
526     </reference>  
527  
528     <property name="currency" type="xsd:string">USD</property>  
529  
530 </componentType>  
531
```

532 **1.4.1.2 Example Implementation**

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

535 **AccountServiceImpl** implements the **AccountService** interface, which is defined via a Java
536 interface:

```
537  
538 package services.account;  
539  
540 @Remotable  
541 public interface AccountService{  
542     public AccountReport getAccountReport(String customerID);  
543 }  
544
```

545

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

```
550  
551 package services.account;  
552  
553 import java.util.List;  
554  
555 import commonj.sdo.DataFactory;  
556  
557 import org.osea.sca.annotations.Property;
```

```

558 import org.osoa.sca.annotations.Reference;
559
560 import services.accountdata.AccountDataService;
561 import services.accountdata.CheckingAccount;
562 import services.accountdata.SavingsAccount;
563 import services.accountdata.StockAccount;
564 import services.stockquote.StockQuoteService;
565
566 public class AccountServiceImpl implements AccountService {
567
568     @Property
569     private String currency = "USD";
570
571     @Reference
572     private AccountDataService accountDataService;
573     @Reference
574     private StockQuoteService stockQuoteService;
575
576     public AccountReport getAccountReport(String customerID) {
577
578         DataFactory dataFactory = DataFactory.INSTANCE;
579         AccountReport accountReport = (AccountReport)dataFactory.create(AccountReport.class);
580         List accountSummaries = accountReport.getAccountSummaries();
581
582         CheckingAccount checkingAccount = accountDataService.getCheckingAccount(customerID);
583         AccountSummary checkingAccountSummary = (AccountSummary)dataFactory.create(AccountSummary.class);
584         checkingAccountSummary.setAccountNumber(checkingAccount.getAccountNumber());
585         checkingAccountSummary.setAccountType("checking");
586         checkingAccountSummary.setBalance(fromUSDollarToCurrency(checkingAccount.getBalance()));
587         accountSummaries.add(checkingAccountSummary);
588
589         SavingsAccount savingsAccount = accountDataService.getSavingsAccount(customerID);
590         AccountSummary savingsAccountSummary = (AccountSummary)dataFactory.create(AccountSummary.class);
591         savingsAccountSummary.setAccountNumber(savingsAccount.getAccountNumber());
592         savingsAccountSummary.setAccountType("savings");
593         savingsAccountSummary.setBalance(fromUSDollarToCurrency(savingsAccount.getBalance()));
594         accountSummaries.add(savingsAccountSummary);
595
596         StockAccount stockAccount = accountDataService.getStockAccount(customerID);
597         AccountSummary stockAccountSummary = (AccountSummary)dataFactory.create(AccountSummary.class);
598         stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());
599         stockAccountSummary.setAccountType("stock");
600         float balance = (stockQuoteService.getQuote(stockAccount.getSymbol()))*stockAccount.getQuantity();
601         stockAccountSummary.setBalance(fromUSDollarToCurrency(balance));
602         accountSummaries.add(stockAccountSummary);
603
604         return accountReport;
605     }
606
607     private float fromUSDollarToCurrency(float value){
608
609         if (currency.equals("USD")) return value; else
610         if (currency.equals("EURO")) return value * 0.8f; else
611         return 0.0f;
612     }
613 }
614

```

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

```

617
618 <?xml version="1.0" encoding="ASCII"?>
619 <componentType xmlns="http://www.osoa.org/xmlns/sca/1.0"
620               xmlns:xsd="http://www.w3.org/2001/XMLSchema">
621
622     <service name="AccountService">
623         <interface.java interface="services.account.AccountService"/>
624     </service>
625     <reference name="accountDataService">
626         <interface.java interface="services.accountdata.AccountDataService"/>
627     </reference>

```

```
628     <reference name="stockQuoteService">
629         <interface.java interface="services.stockquote.StockQuoteService"/>
630     </reference>
631
632     <property name="currency" type="xsd:string">USD</property>
633
634 </componentType>
635
```

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

639

640 **1.5 Interface**

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

647 SCA currently supports the following interface type systems:

- 648 • Java interfaces
- 649 • WSDL 1.1 portTypes
- 650 • WSDL 2.0 interfaces

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

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

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

656

```
657 <interface.java interface="NCName" ... />
658
```

659 The interface.java element has the following attributes:

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

661

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

663

```
664 <interface.java interface="services.stockquote.StockQuoteService"/>
665
```

666

666 Here, the Java interface is defined in the Java class file
667 ./services/stockquote/StockQuoteService.class, where the root directory is defined by the
668 contribution in which the interface exists.

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

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

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

```
676  
677 <interface.wSDL interface="xs:anyURI" ... />
```

678
679 The interface.wSDL element has the following attributes:

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

682
683 The following snippet shows a sample for the WSDL portType/interface element.

```
684  
685 <interface.wSDL interface="http://www.stockquote.org/StockQuoteService#  
686 wsdl.interface(StockQuote)"/>  
687
```

688 For WSDL 1.1, the interface attribute points to a portType in the WSDL. For WSDL 2.0, the
689 interface attribute points to an interface in the WSDL. For the WSDL 1.1 portType and WSDL 2.0
690 interface type systems, arguments and return of the service operations are described using XML
691 schema.

692
693

694 1.5.1 Local and Remotable Interfaces

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

701

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

705
706 Independent of whether the remotable service is called remotely from outside the process where
707 the service runs or from another component running in the same process, the data exchange
708 semantics are **by-value**.

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

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

```
715  
716 package services.hello;  
717  
718 @Remotable
```



```
719 public interface HelloService {
720
721     String hello(String message);
722 }
```

723

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

729

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

734

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

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

738

739 1.5.2 Bidirectional Interfaces

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

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

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

752

```
753 <interface.java          interface="services.invoicing.ComputePrice"
754                        callbackInterface="services.invoicing.InvoiceCallback"/>
```

755

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

759

760 If a reference is defined using a bidirectional interface element, the client component
761 implementation using the reference calls the referenced service using the interface. The client
762 component implementation must implement the callback interface.

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

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

771 1.5.3 Conversational Interfaces

772
773 Services sometimes cannot easily be defined so that each operation stands alone and is
774 completely independent of the other operations of the same service. Instead, there is a
775 sequence of operations that must be called in order to achieve some higher level goal. SCA calls
776 this sequence of operations a **conversation**. If the service uses a bidirectional interface, the
777 conversation may include both operations and callbacks.

778
779 Such conversational services are typically managed by using conversation identifiers that are
780 either (1) part of the application data (message parts or operation parameters) or 2)
781 communicated separately from application data (possibly in headers). SCA introduces the
782 concept of *conversational interfaces* for describing the interface contract for conversational
783 services of the second form above. With this form, it is possible for the runtime to automatically
784 manage the conversation, with the help of an appropriate binding specified at deployment. SCA
785 does not standardize any aspect of conversational services that are maintained using application
786 data. Such services are neither helped nor hindered by SCA's conversational service support.

787
788 Conversational services typically involve state data that relates to the conversation that is taking
789 place. The creation and management of the state data for a conversation has a significant
790 impact on the development of both clients and implementations of conversational services.

791
792 Traditionally, application developers who have needed to write conversational services have been
793 required to write a lot of plumbing code. They need to:

- 794
795 - choose or define a protocol to communicate conversational (correlation) information
796 between the client & provider
- 797 - route conversational messages in the provider to a machine that can handle that
798 conversation, while handling concurrent data access issues
- 799 - write code in the client to use/encode the conversational information
- 800 - maintain state that is specific to the conversation, sometimes persistently and
801 transactionally, both in the implementation and the client.

802
803 SCA makes it possible to divide the effort associated with conversational services between a
804 number of roles:

- 805 - Application Developer: Declares that a service interface is conversational (leaving the
806 details of the protocol up to the binding). Uses lifecycle semantics, APIs or other
807 programmatic mechanisms (as defined by the implementation-type being used) to
808 manage conversational state.
- 809 - Application Assembler: chooses a binding that can support conversations
- 810 - Binding Provider: implements a protocol that can pass conversational information with
811 each operation request/response.

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

816

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

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

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

- 829 - Web service binding with a WS-RM policy
- 830 - Web service binding with a WS-Addressing policy
- 831 - Web service binding with a WS-Context policy
- 832 - JMS binding with a conversation policy that uses the JMS correlationID header

833

834 Conversations occur between one client and one target service. Consequently, requests
835 originating from one client to multiple target conversational services will result in multiple
836 conversations. For example, if a client A calls services B and C, both of which implement
837 conversational interfaces, two conversations result, one between A and B and another between A
838 and C. Likewise, requests flowing through multiple implementation instances will result in
839 multiple conversations. For example, a request flowing from A to B and then from B to C will
840 involve two conversations (A and B, B and C). In the previous example, if a request was then
841 made from C to A, a third conversation would result (and the implementation instance for A
842 would be different from the one making the original request).

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

850

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

858 A `sca:ConversationViolation` fault is thrown when one of the following errors occurs:

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

- 860 - The conversation identification is invalid (not unique, out of range, etc.)
- 861 - The conversation identification is not present in the input message of the operation that
- 862 ends the conversation
- 863 - The client or the service attempts to send a message in a conversation, after the
- 864 conversation has ended

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

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

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

878

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

882

883 1.5.4 SCA-Specific Aspects for WSDL Interfaces

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

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

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

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

896

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

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

```
899 </simpleType>
```

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

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

```
906 requires="conversational"
```

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

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

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

```
915 ...  
916 <portType name="LoanService" sca:requires="conversational">  
917   <operation name="apply">  
918     <input message="tns:ApplicationInput"/>  
919     <output message="tns:ApplicationOutput"/>  
920   </operation>  
921   <operation name="cancel" sca:endsConversation="true">  
922     </operation>  
923   ...  
924 </portType>  
925 ...
```

927 1.6 Composite

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

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

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

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

945
946 A composite is defined in an **xxx.composite** file. A composite is represented by a **composite**
947 element. The following snippet shows the schema for the composite element.

```
948  
949 <?xml version="1.0" encoding="ASCII"?>  
950 <!-- Composite schema snippet -->  
951 <composite xmlns="http://www.oesa.org/xmlns/sca/1.0"  
952   targetNamespace="xs:anyURI"  
953   name="xs:NCName" local="xs:boolean"?  
954   autowire="xs:boolean"? constrainingType="QName"?  
955   requires="list of xs:QName"? policySets="list of xs:QName"?>  
956  
957   <include name="xs:QName"/>*
```

```

958
959 <service name="xs:NCName" promote="xs:anyURI"
960     requires="list of xs:QName"? policySets="list of xs:QName"?>*
961 <interface/>?
962 <binding uri="xs:anyURI"? name="xs:QName"?
963     requires="list of xs:QName"? policySets="list of xs:QName"?/>*
964 <callback>?
965     <binding uri="xs:anyURI"? name="xs:QName"?
966         requires="list of xs:QName"?
967         policySets="list of xs:QName"?/>+
968 </callback>
969 </service>
970
971 <reference name="xs:NCName" target="list of xs:anyURI"?
972     promote="list of xs:anyURI" wiredByImpl="xs:boolean"?
973     multiplicity="0..1 or 1..1 or 0..n or 1..n"?
974     requires="list of xs:QName"? policySets="list of xs:QName"?>*
975 <interface/>?
976 <binding uri="xs:anyURI"? name="xs:QName"?
977     requires="list of xs:QName"? policySets="list of xs:QName"?/>*
978 <callback>?
979     <binding uri="xs:anyURI"? name="xs:QName"?
980         requires="list of xs:QName"?
981         policySets="list of xs:QName"?/>+
982 </callback>
983 </reference>
984
985 <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
986     many="xs:boolean"? mustSupply="xs:boolean"?>*
987     default-property-value?
988 </property>
989
990 <component name="xs:NCName" autowire="xs:boolean"?
991     requires="list of xs:QName"? policySets="list of xs:QName"?>*
992 <implementation/>?
993 <service name="xs:NCName" requires="list of xs:QName"?
994     policySets="list of xs:QName"?>*
995 <interface/>?
996 <binding uri="xs:anyURI"? name="xs:QName"?
997     requires="list of xs:QName"?
998     policySets="list of xs:QName"?/>*
999 <callback>?
1000     <binding uri="xs:anyURI"? name="xs:QName"?
1001         requires="list of xs:QName"?
1002         policySets="list of xs:QName"?/>+
1003 </callback>
1004 </service>
1005 <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
1006     source="xs:string"? file="xs:anyURI"?>*
1007     property-value
1008 </property>
1009 <reference name="xs:NCName" target="list of xs:anyURI"?
1010     autowire="xs:boolean"? wiredByImpl="xs:boolean"?
1011     requires="list of xs:QName"? policySets="list of xs:QName"?
1012     multiplicity="0..1 or 1..1 or 0..n or 1..n"?/>*
1013 <interface/>?
1014 <binding uri="xs:anyURI"? name="xs:QName"?
1015     requires="list of xs:QName"?
1016     policySets="list of xs:QName"?/>*
1017 <callback>?

```

```

1018         <binding uri="xs:anyURI"? name="xs:QName"?
1019             requires="list of xs:QName"?
1020             policySets="list of xs:QName"?/>+
1021     </callback>
1022 </reference>
1023 </component>
1024
1025 <wire source="xs:anyURI" target="xs:anyURI" />*
1026
1027 </composite>
1028
1029
1030

```

The composite element has the following *attributes*:

- 1031 • **name (required)** – the name of the composite. The form of a composite name is an XML
1032 QName, in the namespace identified by the targetNamespace attribute.
- 1033 • **targetNamespace (optional)** – an identifier for a target namespace into which the
1034 composite is declared
- 1035 • **local (optional)** – whether all the components within the composite must all run in the
1036 same operating system process. local="true" means that all the components must run in
1037 the same process. local="false", which is the default, means that different components
1038 within the composite may run in different operating system processes and they may even
1039 run on different nodes on a network.
- 1040 • **autowire (optional)** – whether contained component references should be autowired, as
1041 described in [the Autowire section](#). Default is false.
- 1042 • **constrainingType (optional)** – the name of a constrainingType. When specified, the
1043 set of services, references and properties of the composite, plus related intents, is
1044 constrained to the set defined by the constrainingType. See [the ConstrainingType Section](#)
1045 for more details.
- 1046 • **requires (optional)** – a list of policy intents. See the [Policy Framework specification](#)
1047 [\[10\]](#) for a description of this attribute.
- 1048 • **policySets (optional)** – a list of policy sets. See the [Policy Framework specification \[10\]](#)
1049 for a description of this attribute.

1050 Composites contain **zero or more properties, services, components, references, wires and**
1051 **included composites**. These artifacts are described in detail in the following sections.

1052 Components contain configured implementations which hold the business logic of the composite.
1053 The components offer services and require references to other services. Composite services
1054 define the public services provided by the composite, which can be accessed from outside the
1055 composite. Composite references represent dependencies which the composite has on services
1056 provided elsewhere, outside the composite. Wires describe the connections between component
1057 services and component references within the composite. Included composites contribute the
1058 elements they contain to the using composite.

1059 Composite services involve the **promotion** of one service of one of the components within the
1060 composite, which means that the composite service is actually provided by one of the
1061 components within the composite. Composite references involve the **promotion** of one or more
1062 references of one or more components. Multiple component references can be promoted to the
1063 same composite reference, as long as all the component references are compatible with one
1064 another. Where multiple component references are promoted to the same composite reference,
1065 then they all share the same configuration, including the same target service(s).

1066 Composite services and composite references can use the configuration of their promoted
1067 services and references respectively (such as Bindings and Policy Sets). Alternatively composite
1068 services and composite references can override some or all of the configuration of the promoted

1069 services and references, through the configuration of bindings and other aspects of the
1070 composite service or reference.

1071 Component services and component references can be promoted to composite services and
1072 references and also be wired internally within the composite at the same time. For a reference,
1073 this only makes sense if the reference supports a multiplicity greater than 1.

1074 1.6.1 Property – Definition and Configuration

1075 **Properties** allow for the configuration of an implementation with externally set data values. An
1076 implementation, including a composite, can declare zero or more properties. Each property has
1077 a type, which may be either simple or complex. An implementation may also define a default
1078 value for a property. Properties are configured with values in the components that use the
1079 implementation.

1080 The declaration of a property in a composite follows the form described in the following schema
1081 snippet:

1082

```
1083 <?xml version="1.0" encoding="ASCII"?>  
1084  
1085 <composite xmlns="http://www.oesa.org/xmlns/sca/1.0"  
1086           name="xs:QName" ... >  
1087  
1088     ...  
1089  
1090     <property name="xs:NCName" (type="xs:QName" | element="xs:QName")  
1091             many="xs:boolean"? mustSupply="xs:boolean"?> *  
1092             default-property-value?  
1093     </property>  
1094     ...  
1095  
1096 </composite>  
1097
```

1098 The property element has the following attributes:

- 1099 ▪ **name (required)** - the name of the property
- 1100 ▪ one of **(required)**:
 - 1101 ○ **type** – the type of the property - the qualified name of an XML schema type
 - 1102 ○ **element** – the type of the property defined as the qualified name of an XML
1103 schema global element – the type is the type of the global element
- 1104 ▪ **many (optional)** - whether the property is single-valued (false) or multi-valued (true).
1105 The default is **false**. In the case of a multi-valued property, it is presented to the
1106 implementation as a Collection of property values.
- 1107 ▪ **mustSupply (optional)** – whether the property value must be supplied by the
1108 component that uses the implementation – when mustSupply="true" the component must
1109 supply a value since the implementation has no default value for the property. A default-
1110 property-value should only be supplied when mustSupply="false" (the default setting for
1111 the mustSupply attribute), since the implication of a default value is that it is used only
1112 when a value is not supplied by the using component.

1113 The property element may contain an optional **default-property-value**, which provides default
1114 value for the property. The default value must match the type declared for the property:

- 1115 ○ a string, if **type** is a simple type (must match the **type** declared)
- 1116 ○ a complex type value matching the type declared by **type**
- 1117 ○ an element matching the element named by **element**
- 1118 ○ multiple values are permitted if many="true" is specified

1119

1120 Implementation types other than **composite** can declare properties in an implementation-
1121 dependent form (eg annotations within a Java class), or through a property declaration of exactly
1122 the form described above in a componentType file.

1123 Property values can be configured when an implementation is used by a component. The form of
1124 the property configuration is shown in [the section on Components](#).

1125 1.6.1.1 Property Examples

1126

1127 For the following example of Property declaration and value setting, the following complex type is
1128 used as an example:

```
1129 <xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"  
1130             targetNamespace="http://foo.com/"  
1131             xmlns:tns="http://foo.com/">  
1132   <!-- ComplexProperty schema -->  
1133   <xsd:element name="fooElement" type="MyComplexType"/>  
1134   <xsd:complexType name="MyComplexType">  
1135     <xsd:sequence>  
1136       <xsd:element name="a" type="xsd:string"/>  
1137       <xsd:element name="b" type="anyURI"/>  
1138     </xsd:sequence>  
1139     <attribute name="attr" type="xsd:string" use="optional"/>  
1140   </xsd:complexType>  
1141 </xsd:schema>
```

1142

1143 The following composite demonstrates the declaration of a property of a complex type, with a
1144 default value, plus it demonstrates the setting of a property value of a complex type within a
1145 component:

```
1146 <?xml version="1.0" encoding="ASCII"?>  
1147  
1148 <composite xmlns="http://www.oesa.org/xmlns/sca/1.0"  
1149            xmlns:foo="http://foo.com"  
1150            targetNamespace="http://foo.com"  
1151            name="AccountServices">  
1152 <!-- AccountServices Example1 -->  
1153  
1154   ...  
1155  
1156   <property name="complexFoo" type="foo:MyComplexType">  
1157     <MyComplexPropertyValue xsi:type="foo:MyComplexType">  
1158       <foo:a>AValue</foo:a>  
1159       <foo:b>InterestingURI</foo:b>  
1160     </MyComplexPropertyValue>  
1161   </property>  
1162  
1163   <component name="AccountServiceComponent">  
1164     <implementation.java class="foo.AccountServiceImpl"/>  
1165     <property name="complexBar" source="$complexFoo"/>  
1166     <reference name="accountDataService"  
1167               target="AccountDataServiceComponent"/>  
1168     <reference name="stockQuoteService" target="StockQuoteService"/>  
1169   </component>  
1170  
1171   ...  
1172 </composite>
```

1173

1174 In the declaration of the property named **complexFoo** in the composite **AccountServices**, the
1175 property is defined to be of type **foo:MyComplexType**. The namespace **foo** is declared in the
1176 composite and it references the example XSD, where MyComplexType is defined. The
1177 declaration of complexFoo contains a default value. This is declared as the content of the
1178 property element. In this example, the default value consists of the element
1179 **MyComplexPropertyValue** of type foo:MyComplexType and its two child elements <foo:a> and
1180 <foo:b>, following the definition of MyComplexType.

1181 In the component **AccountServiceComponent**, the component sets the value of the property
1182 **complexBar**, declared by the implementation configured by the component. In this case, the
1183 type of complexBar is foo:MyComplexType. The example shows that the value of the
1184 complexBar property is set from the value of the complexFoo property – the **source** attribute of
1185 the property element for complexBar declares that the value of the property is set from the value
1186 of a property of the containing composite. The value of the source attribute is **\$complexFoo**,
1187 where complexFoo is the name of a property of the composite. This value implies that the whole
1188 of the value of the source property is used to set the value of the component property.

1189 The following example illustrates the setting of the value of a property of a simple type (a string)
1190 from **part** of the value of a property of the containing composite which has a complex type:

```
1191 <?xml version="1.0" encoding="ASCII"?>
1192
1193 <composite      xmlns="http://www.osea.org/xmlns/sca/1.0"
1194                xmlns:foo="http://foo.com"
1195                targetNamespace="http://foo.com"
1196                name="AccountServices">
1197 <!-- AccountServices Example2 -->
1198
1199     ...
1200
1201     <property name="complexFoo" type="foo:MyComplexType">
1202         <MyComplexPropertyValue xsi:type="foo:MyComplexType">
1203             <foo:a>AValue</foo:a>
1204             <foo:b>InterestingURI</foo:b>
1205         </MyComplexPropertyValue>
1206     </property>
1207
1208     <component name="AccountServiceComponent">
1209         <implementation.java class="foo.AccountServiceImpl"/>
1210         <property name="currency" source="$complexFoo/a"/>
1211         <reference name="accountDataService"
1212                 target="AccountDataServiceComponent"/>
1213         <reference name="stockQuoteService" target="StockQuoteService"/>
1214     </component>
1215
1216     ...
1217
1218 </composite>
```

1219 In this example, the component **AccountServiceComponent** sets the value of a property called
1220 **currency**, which is of type string. The value is set from a property of the composite
1221 **AccountServices** using the source attribute set to **\$complexFoo/a**. This is an XPath
1222 expression that selects the property name **complexFoo** and then selects the value of the **a**
1223 subelement of complexFoo. The "a" subelement is a string, matching the type of the currency
1224 property.

1225 Further examples of declaring properties and setting property values in a component follow:

1226 Declaration of a property with a simple type and a default value:

```
1227 <property name="SimpleTypeProperty" type="xsd:string">
1228 MyValue
```

1229 </property>

1230

1231 Declaration of a property with a complex type and a default value:

```
1232 <property name="complexFoo" type="foo:MyComplexType">
1233   <MyComplexPropertyValue xsi:type="foo:MyComplexType">
1234     <foo:a>AValue</foo:a>
1235     <foo:b>InterestingURI</foo:b>
1236   </MyComplexPropertyValue>
1237 </property>
```

1238

1239 Declaration of a property with an element type:

```
1240 <property name="elementFoo" element="foo:fooElement">
1241   <foo:fooElement>
1242     <foo:a>AValue</foo:a>
1243     <foo:b>InterestingURI</foo:b>
1244   </foo:fooElement>
1245 </property>
```

1246

1247 Property value for a simple type:

```
1248 <property name="SimpleTypeProperty">
1249   MyValue
1250 </property>
```

1251

1252

1253 Property value for a complex type, also showing the setting of an attribute value of the complex
1254 type:

```
1255 <property name="complexFoo">
1256   <MyComplexPropertyValue xsi:type="foo:MyComplexType" attr="bar">
1257     <foo:a>AValue</foo:a>
1258     <foo:b>InterestingURI</foo:b>
1259   </MyComplexPropertyValue>
1260 </property>
```

1261

1262 Property value for an element type:

```
1263 <property name="elementFoo">
1264   <foo:fooElement attr="bar">
1265     <foo:a>AValue</foo:a>
1266     <foo:b>InterestingURI</foo:b>
1267   </foo:fooElement>
1268 </property>
```

1269

1270 Declaration of a property with a complex type where multiple values are supported:

```
1271 <property name="complexFoo" type="foo:MyComplexType" many="true"/>
```

1272

1273 Setting of a value for that property where multiple values are supplied:

```
1274 <property name="complexFoo">
1275   <MyComplexPropertyValue1 xsi:type="foo:MyComplexType" attr="bar">
1276     <foo:a>AValue</foo:a>
```

```

1277     <foo:b>InterestingURI</foo:b>
1278 </MyComplexPropertyValue1>
1279 <MyComplexPropertyValue2 xsi:type="foo:MyComplexType" attr="zing">
1280     <foo:a>BValue</foo:a>
1281     <foo:b>BoringURI</foo:b>
1282 </MyComplexPropertyValue2>
1283 </property>
1284
1285

```

1.6.2 References

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

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

```

1294
1295 <?xml version="1.0" encoding="ASCII"?>
1296 <!-- Reference schema snippet -->
1297 <composite xmlns="http://www.osea.org/xmlns/sca/1.0"
1298     targetNamespace="xs:anyURI"
1299     name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?
1300     constrainingType="QName"?
1301     requires="list of xs:QName"? policySets="list of xs:QName"?>
1302
1303     ...
1304
1305     <reference name="xs:NCName" target="list of xs:anyURI"?
1306         promote="list of xs:anyURI" wiredByImpl="xs:boolean"?
1307         multiplicity="0..1 or 1..1 or 0..n or 1..n"?
1308         requires="list of xs:QName"? policySets="list of xs:QName"?>*
1309     <interface/>?
1310     <binding uri="xs:anyURI"? name="xs:QName"?
1311         requires="list of xs:QName" policySets="list of xs:QName"?/>*
1312     <callback?>
1313         <binding uri="xs:anyURI"? name="xs:QName"?
1314             requires="list of xs:QName"?
1315             policySets="list of xs:QName"?/>+
1316     </callback>
1317 </reference>
1318
1319     ...
1320
1321 </composite>
1322
1323

```

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

- ***name (required)*** – the name of the reference. The name must be unique across all the composite references in the composite. The name of the composite reference can be different then the name of the promoted component reference.
- ***promote (required)*** – identifies one or more promoted component references. The value is a list of values of the form <component-name>/<reference-name> separated by

- spaces. The specification of the reference name is optional if the component has only one reference.
- **requires (optional)** – a list of required policy intents. See the [Policy Framework specification \[10\]](#) for a description of this attribute.
 - **policySets (optional)** – a list of policy sets. See the [Policy Framework specification \[10\]](#) for a description of this attribute.
 - **multiplicity (optional)** - Defines the number of wires that can connect the reference to target services. The multiplicity can have the following values
 - 1..1 – one wire can have the reference as a source
 - 0..1 – zero or one wire can have the reference as a source
 - 1..n – one or more wires can have the reference as a source
 - 0..n - zero or more wires can have the reference as a source
 - **target (optional)** – a list of one or more of target service URI's, depending on multiplicity setting. Each value wires the reference to a service in a composite that uses the composite containing the reference as an implementation for one of its components. For more details on wiring see [the section on Wires](#).
 - **wiredByImpl (optional)** – a boolean value, "false" by default, which indicates that the implementation wires this reference dynamically. If set to "true" it indicates that the target of the reference is set at runtime by the implementation code (eg by the code obtaining an endpoint reference by some means and setting this as the target of the reference through the use of programming interfaces defined by the relevant Client and Implementation specification). If "true" is set, then the reference should not be wired statically within a using composite, but left unwired.

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

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

The value specified for the **multiplicity** attribute has to be compatible with the multiplicity specified on the component reference, i.e. it has to be equal or further restrict. So a composite reference of multiplicity 0..1 or 1..1 can be used where the promoted component reference has multiplicity 0..n and 1..n respectively. 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.

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

If one or more **bindings** are specified they **override** any and all of the bindings defined for the promoted component reference from the composite reference perspective. The bindings defined on the component reference are still in effect for local wires within the composite that have the component reference as their source. A reference element has zero or more **binding elements**

1377 as children. Details of the binding element are described in the [Bindings section](#). For more details
1378 on wiring see [the section on Wires](#).

1379 Note that a binding element may specify an endpoint which is the target of that binding. A
1380 reference must not mix the use of endpoints specified via binding elements with target endpoints
1381 specified via the target attribute. If the target attribute is set, then binding elements can only
1382 list one or more binding types that can be used for the wires identified by the target attribute.
1383 All the binding types identified are available for use on each wire in this case. If endpoints are
1384 specified in the binding elements, each endpoint must use the binding type of the binding
1385 element in which it is defined. In addition, each binding element needs to specify an endpoint in
1386 this case.

1387 A **reference** element has an optional **callback** element used if the interface has a callback
1388 defined, which has one or more **binding** elements as children. The **callback** and its binding
1389 child elements are specified if there is a need to have binding details used to handle callbacks. If
1390 the callback element is not present, the behaviour is runtime implementation dependent.

1391

1392 The same component reference maybe promoted more than once, using different composite
1393 references, but only if the multiplicity defined on the component reference is 0..n or 1..n. The
1394 multiplicity on the composite reference can restrict accordingly.

1395 Two or more component references may be promoted by one composite reference, but only
1396 when

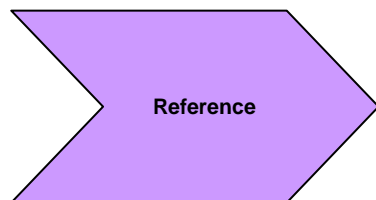
- 1397 • the interfaces of the component references are the same, or if the composite reference
1398 itself declares an interface then all the component references must have interfaces which
1399 are compatible with the composite reference interface
- 1400 • the multiplicities of the component references are compatible, i.e one can be the
1401 restricted form of the another, which also means that the composite reference carries the
1402 restricted form either implicitly or explicitly
- 1403 • the intents declared on the component references must be compatible – the intents which
1404 apply to the composite reference in this case are the union of the required intents
1405 specified for each of the promoted component references. If any intents contradict (eg
1406 mutually incompatible qualifiers for a particular intent) then there is an error.

1407

1408 **1.6.2.1 Example Reference**

1409

1410 The following figure shows the reference symbol that is used to represent a reference in an
1411 assembly diagram.

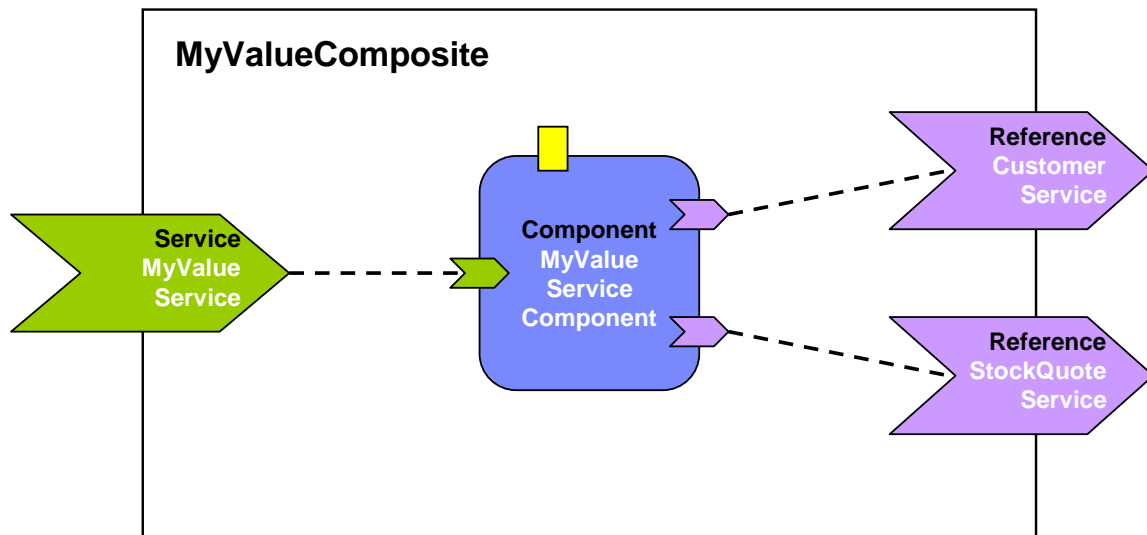


1412

1413 **Figure 7: Reference symbol**

1414 The following figure shows the assembly diagram for the MyValueComposite containing the
1415 reference CustomerService and the reference StockQuoteService.

1416



1417
1418 **Figure 8: MyValueComposite showing References**
1419

1420 The following snippet shows the MyValueComposite.composite file for the MyValueComposite
1421 containing the reference elements for the CustomerService and the StockQuoteService. The
1422 reference CustomerService is bound using the SCA binding. The reference StockQuoteService is
1423 bound using the Web service binding. The endpoint addresses of the bindings can be specified,
1424 for example using the binding *uri* attribute (for details see the [Bindings](#) section), or overridden in
1425 an enclosing composite. Although in this case the reference StockQuoteService is bound to a
1426 Web service, its interface is defined by a Java interface, which was created from the WSDL
1427 portType of the target web service.

1428
1429

```
<?xml version="1.0" encoding="ASCII"?>
1430 <!-- MyValueComposite_3 example -->
1431 <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
1432           targetNamespace="http://foo.com"
1433           name="MyValueComposite" >
1434
1435   ...
1436
1437   <component name="MyValueServiceComponent">
1438     <implementation.java class="services.myvalue.MyValueServiceImpl"/>
1439     <property name="currency">EURO</property>
1440     <reference name="customerService"/>
1441     <reference name="StockQuoteService"/>
1442   </component>
1443
1444   <reference name="CustomerService"
1445     promote="MyValueServiceComponent/customerService">
1446     <interface.java interface="services.customer.CustomerService"/>
1447     <!-- The following forces the binding to be binding.sca whatever is -->
1448     <!-- specified by the component reference or by the underlying -->
1449     <!-- implementation -->
1450     <binding.sca/>
1451   </reference>
1452
1453   <reference name="StockQuoteService"
1454     promote="MyValueServiceComponent/StockQuoteService">
1455     <interface.java interface="services.stockquote.StockQuoteService"/>
1456     <binding.ws port="http://www.stockquote.org/StockQuoteService#
```

```

1457         wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
1458     </reference>
1459     ...
1461 </composite>
1462
1463

```

1.6.3 Service

1464 The **services of a composite** are defined by promoting services defined by components
 1465 contained in the composite. A component service is promoted by means of a composite **service**
 1466 **element**.

1468 A composite service is represented by a **service element** which is a child of the composite
 1469 element. There can be **zero or more** service elements in a composite. The following snippet
 1470 shows the composite schema with the schema for a service child element:

```

1471
1472 <?xml version="1.0" encoding="ASCII"?>
1473 <!-- Serviceee schema snippet -->
1474 <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
1475           targetNamespace="xs:anyURI"
1476           name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?
1477           constrainingType="QName"?
1478           requires="list of xs:QName"? policySets="list of xs:QName"?>
1479     ...
1480
1481     <service name="xs:NCName" promote="xs:anyURI"
1482           requires="list of xs:QName"? policySets="list of xs:QName"?>*
1483       <interface/>?
1484       <binding uri="xs:anyURI"? name="xs:QName"?
1485             requires="list of xs:QName" policySets="list of xs:QName"?/>*
1486       <callback?
1487         <binding uri="xs:anyURI"? name="xs:QName"?
1488               requires="list of xs:QName"?
1489               policySets="list of xs:QName"?/>+
1490     </callback>
1491   </service>
1492   ...
1493 </composite>
1494
1495
1496

```

1497 The service element has the following **attributes**:

- 1498 • **name (required)** – the name of the service, the name MUST BE unique across all the
 1499 composite services in the composite. The name of the composite service can be different
 1500 from the name of the promoted component service.
- 1501 • **promote (required)** – identifies the promoted service, the value is of the form
 1502 <component-name>/<service-name>. The service name is optional if the target
 1503 component only has one service.
- 1504 • **requires (optional)** – a list of required policy intents. See the [Policy Framework](#)
 1505 [specification \[10\]](#) for a description of this attribute.
- 1506 • **policySets (optional)** – a list of policy sets. See the [Policy Framework specification \[10\]](#)
 1507 for a description of this attribute.

1508

1509 The composite service can optionally specify an **interface**, **required intents** and **bindings**.
1510 Whatever is not specified is defaulted from the promoted component service.

1511

1512 If an **interface** is specified it must be the same or a compatible subset of the interface provided
1513 by the promoted component service, i.e. provide a subset of the operations defined by the
1514 component service. The interface is described by **zero or one interface element** which is a
1515 child element of the service element. For details on the interface element see [the Interface](#)
1516 [section](#).

1517

1518 Specified **required intents** add to or further qualify the required intents defined by the
1519 promoted component service.

1520

1521 If bindings are specified they **override** the bindings defined for the promoted component service
1522 from the composite service perspective. The bindings defined on the component service are still
1523 in effect for local wires within the composite that target the component service. A service
1524 element has zero or more **binding elements** as children. Details of the binding element are
1525 described in the [Bindings section](#). For more details on wiring see [the Wiring section](#).

1526 A service element has an optional **callback** element used if the interface has a callback defined,,
1527 which has one or more **binding** elements as children. The **callback** and its binding child
1528 elements are specified if there is a need to have binding details used to handle callbacks. If the
1529 callback element is not present, the behaviour is runtime implementation dependent.

1530

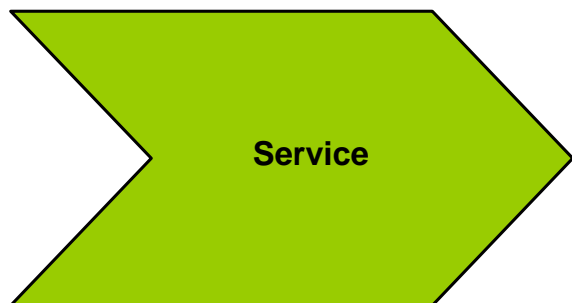
1531 The same component service can be promoted by more then one composite service.

1532

1533 **1.6.3.1 Service Examples**

1534

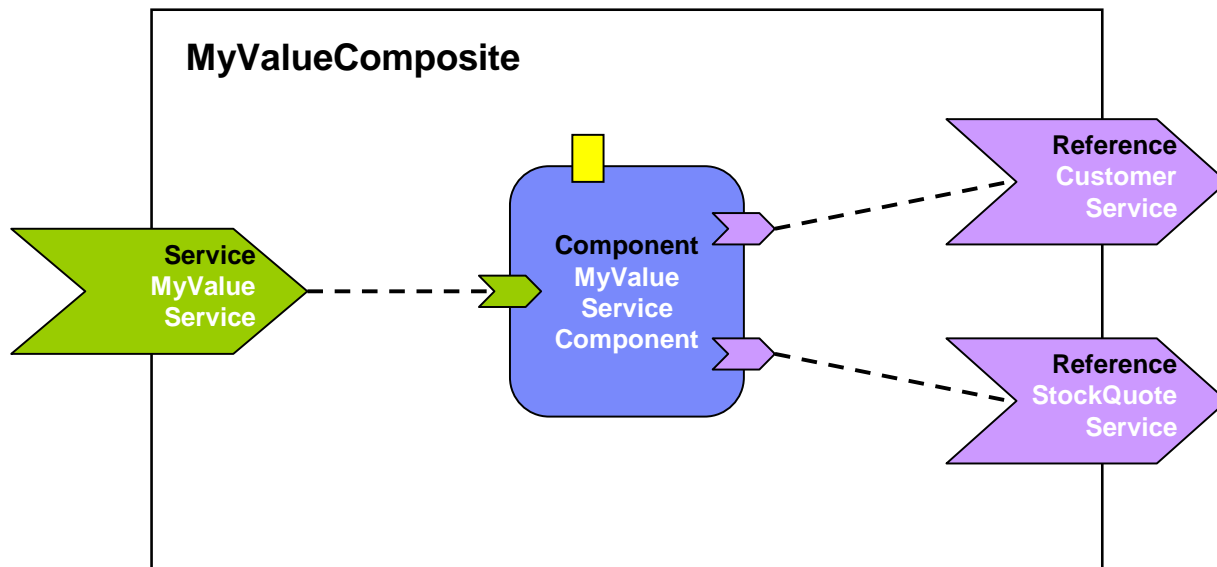
1535 The following figure shows the service symbol that used to represent a service in an assembly
1536 diagram:



1537

1538 **Figure 9: Service symbol**

1539 The following figure shows the assembly diagram for the MyValueComposite containing the
1540 service MyValueService.



1541
1542 **Figure 10: MyValueComposite showing Service**
1543

1544 The following snippet shows the MyValueComposite.composite file for the MyValueComposite
1545 containing the service element for the MyValueService, which is a promote of the service offered
1546 by the MyValueServiceComponent. The name of the promoted service is omitted since
1547 MyValueServiceComponent offers only one service. The composite service MyValueService is
1548 bound using a Web service binding.

1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575

```

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

```

1576 1.6.4 Wire

1577 **SCA wires** within a composite connect **source component references** to **target component**
1578 **services**.

1579 One way of defining a wire is by **configuring a reference of a component using its target**
1580 **attribute**. The reference element is configured with the wire-target-URI of the service(s) that
1581 resolve the reference. Multiple target services are valid when the reference has a multiplicity of
1582 0..n or 1..n.

1583 An alternative way of defining a Wire is by means of a **wire element** which is a child of the
1584 composite element. There can be **zero or more** wire elements in a composite. This alternative
1585 method for defining wires is useful in circumstances where separation of the wiring from the
1586 elements the wires connect helps simplify development or operational activities. An example is
1587 where the components used to build a domain are relatively static but where new or changed
1588 applications are created regularly from those components, through the creation of new
1589 assemblies with different wiring. Deploying the wiring separately from the components allows
1590 the wiring to be created or modified with minimum effort.

1591 Note that a Wire specified via a wire element is equivalent to a wire specified via the target
1592 attribute of a reference. The rule which forbids mixing of wires specified with the target attribute
1593 with the specification of endpoints in binding subelements of the reference also applies to wires
1594 specified via separate wire elements.

1595 The following snippet shows the composite schema with the schema for the reference elements
1596 of components and composite services and the wire child element:

1597

```
1598 <?xml version="1.0" encoding="ASCII"?>  
1599 <!-- Wires schema snippet -->  
1600 <composite xmlns="http://www.oesa.org/xmlns/sca/1.0"  
1601           targetNamespace="xs:anyURI"  
1602           name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?  
1603           constrainingType="QName"?  
1604           requires="list of xs:QName"? policySets="list of xs:QName"?>  
1605  
1606     ...  
1607  
1608     <wire source="xs:anyURI" target="xs:anyURI" /*>  
1609  
1610 </composite>
```

1611
1612

1613 The **reference element of a component** and the **reference element of a service** has a list
1614 of one or more of the following **wire-target-URI** values for the target, with multiple values
1615 separated by a space:

- 1616 • `<component-name>/<service-name>`
 - 1617 ○ where the target is a service of a component. The specification of the service name
1618 is optional if the target component only has one service with a compatible interface

1619

1620 The **wire element** has the following attributes:

- 1621 • **source (required)** – names the source component reference. Valid URI schemes are:
 - 1622 ○ `<component-name>/<reference-name>`
 - 1623 ▪ where the source is a component reference. The specification of the
1624 reference name is optional if the source component only has one reference
- 1625 • **target (required)** – names the target component service. Valid URI schemes are
 - 1626 ○ `<component-name>/<service-name>`

- 1627 ▪ where the target is a service of a component. The specification of the
1628 service name is optional if the target component only has one service with
1629 a compatible interface

1630 For a composite used as a component implementation, wires can only link sources and targets
1631 that are contained in the same composite (irrespective of which file or files are used to describe
1632 the composite). Wiring to entities outside the composite is done through services and references
1633 of the composite with wiring defined by the next higher composite.

1634 A wire may only connect a source to a target if the target implements an interface that is
1635 compatible with the interface required by the source. The source and the target are compatible
1636 if:

- 1637 1. the source interface and the target interface MUST either both be remotable or they are
1638 both local
- 1639 2. the operations on the target interface MUST be the same as or be a superset of the
1640 operations in the interface specified on the source
- 1641 3. compatibility for the individual operation is defined as compatibility of the signature, that
1642 is operation name, input types, and output types MUST BE the same.
- 1643 4. the order of the input and output types also MUST BE the same.
- 1644 5. the set of Faults and Exceptions expected by the source MUST BE the same or be a
1645 superset of those specified by the target.
- 1646 6. other specified attributes of the two interfaces MUST match, including Scope and Callback
1647 interface

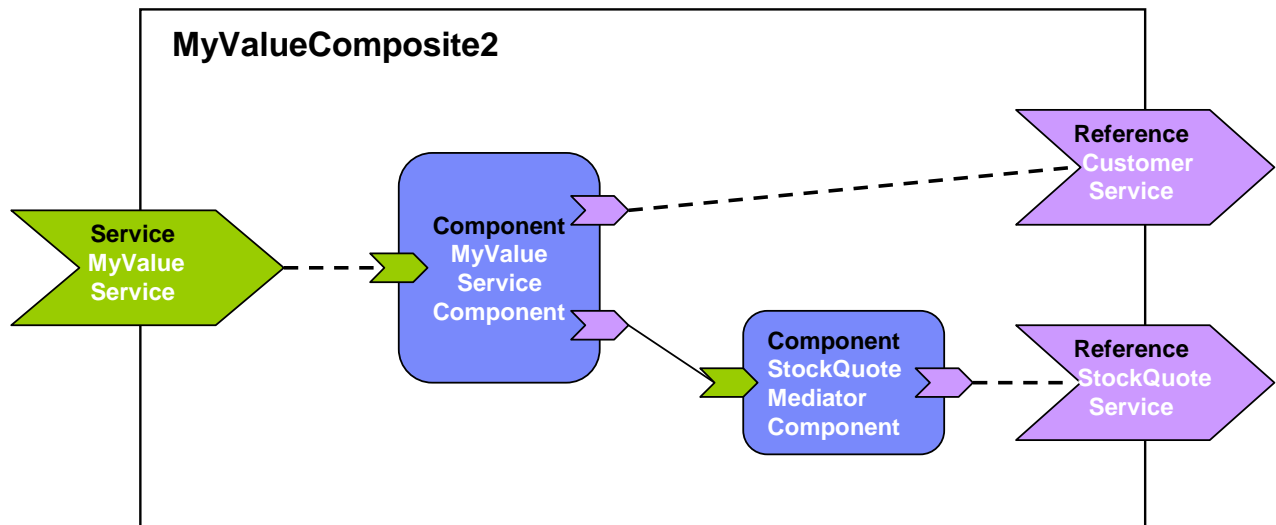
1648 A Wire can connect between different interface languages (eg. Java interfaces and WSDL
1649 portTypes) in either direction, as long as the operations defined by the two interface types are
1650 equivalent. They are equivalent if the operation(s), parameter(s), return value(s) and
1651 faults/exceptions map to each other.

1652 Service clients cannot (portably) ask questions at runtime about additional interfaces that are
1653 provided by the implementation of the service (e.g. the result of “instance of” in Java is non
1654 portable). It is valid for an SCA implementation to have proxies for all wires, so that, for
1655 example, a reference object passed to an implementation may only have the business interface
1656 of the reference and may not be an instance of the (Java) class which is used to implement the
1657 target service, even where the interface is local and the target service is running in the same
1658 process.

1659 **Note:** It is permitted to deploy a composite that has references that are not wired. For the case
1660 of an un-wired reference with multiplicity 1..1 or 1..n the deployment process provided by an
1661 SCA runtime SHOULD issue a warning.

1662 1663 **1.6.4.1 Wire Examples** 1664

1665 The following figure shows the assembly diagram for the MyValueComposite2 containing wires
1666 between service, components and references.



1667
1668 **Figure 11: MyValueComposite2 showing Wires**
1669

1670 The following snippet shows the MyValueComposite2.composite file for the MyValueComposite2
1671 containing the configured component and service references. The service MyValueService is
1672 wired to the MyValueServiceComponent. The MyValueServiceComponent's customerService
1673 reference is wired to the composite's CustomerService reference. The
1674 MyValueServiceComponent's stockQuoteService reference is wired to the
1675 StockQuoteMediatorComponent, which in turn has its reference wired to the StockQuoteService
1676 reference of the composite.

1677

```

1678 <?xml version="1.0" encoding="ASCII"?>
1679 <!-- MyValueComposite Wires examples -->
1680 <composite xmlns="http://www.oesa.org/xmlns/sca/1.0"
1681           targetNamespace="http://foo.com"
1682           name="MyValueComposite2" >
1683
1684     <service name="MyValueService" promote="MyValueServiceComponent">
1685       <interface.java interface="services.myvalue.MyValueService"/>
1686       <binding.ws port="http://www.myvalue.org/MyValueService#
1687         wsdl.endpoint(MyValueService/MyValueServiceSOAP)"/>
1688     </service>
1689
1690     <component name="MyValueServiceComponent">
1691       <implementation.java class="services.myvalue.MyValueServiceImpl"/>
1692       <property name="currency">EURO</property>
1693       <service name="MyValueService"/>
1694       <reference name="customerService"/>
1695       <reference name="stockQuoteService"
1696         target="StockQuoteMediatorComponent"/>
1697     </component>
1698
1699     <component name="StockQuoteMediatorComponent">
1700       <implementation.java class="services.myvalue.SQMediatorImpl"/>
1701       <property name="currency">EURO</property>
1702       <reference name="stockQuoteService"/>
1703     </component>
1704
1705     <reference name="CustomerService"
1706       promote="MyValueServiceComponent/customerService">
1707       <interface.java interface="services.customer.CustomerService"/>

```

```

1708         <binding.sca/>
1709     </reference>
1710
1711     <reference name="StockQuoteService" promote="StockQuoteMediatorComponent">
1712         <interface.java interface="services.stockquote.StockQuoteService"/>
1713         <binding.ws port="http://www.stockquote.org/StockQuoteService#
1714             wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
1715     </reference>
1716
1717 </composite>
1718

```

1719 **1.6.4.2 Autowire**

1720 SCA provides a feature named **Autowire**, which can help to simplify the assembly of composites.
 1721 Autowire enables component references to be automatically wired to component services which
 1722 will satisfy those references, without the need to create explicit wires between the references
 1723 and the services. When the autowire feature is used, a component reference which is not
 1724 promoted and which is not explicitly wired to a service within a composite is automatically wired
 1725 to a target service within the same composite. Autowire works by searching within the
 1726 composite for a service interface which matches the interface of the references.

1727 The autowire feature is not used by default. Autowire is enabled by the setting of an autowire
 1728 attribute to "true". Autowire is disabled by setting of the autowire attribute to "false" The
 1729 autowire attribute can be applied to any of the following elements within a composite:

- 1730 • reference
- 1731 • component
- 1732 • composite

1733 Where an element does not have an explicit setting for the autowire attribute, it inherits the
 1734 setting from its parent element. Thus a reference element inherits the setting from its containing
 1735 component. A component element inherits the setting from its containing composite. Where
 1736 there is no setting on any level, autowire="false" is the default.

1737 As an example, if a composite element has autowire="true" set, this means that autowiring is
 1738 enabled for all component references within that composite. In this example, autowiring can be
 1739 turned off for specific components and specific references through setting autowire="false" on
 1740 the components and references concerned.

1741 For each component reference for which autowire is enabled, the autowire process searches
 1742 within the composite for target services which are compatible with the reference. "Compatible"
 1743 here means:

- 1744 • the target service interface must be a compatible superset of the reference interface (as
 1745 defined in [the section on Wires](#))
- 1746 • the intents, bindings and policies applied to the service must be compatible on the
 1747 reference – so that wiring the reference to the service will not cause an error due to
 1748 binding and policy mismatch (see [the Policy Framework specification \[10\]](#) for details)

1749 If the search finds **more than 1** valid target service for a particular reference, the action taken
 1750 depends on the multiplicity of the reference:

- 1751 • for multiplicity 0..1 and 1..1, the SCA runtime selects one of the target services in a
 1752 runtime-dependent fashion and wires the reference to that target service
- 1753 • for multiplicity 0..n and 1..n, the reference is wired to all of the target services

1754 If the search finds **no** valid target services for a particular reference, the action taken depends
 1755 on the multiplicity of the reference:

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

1.6.4.3 Autowire Examples

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

First, here is a diagram for the composite:

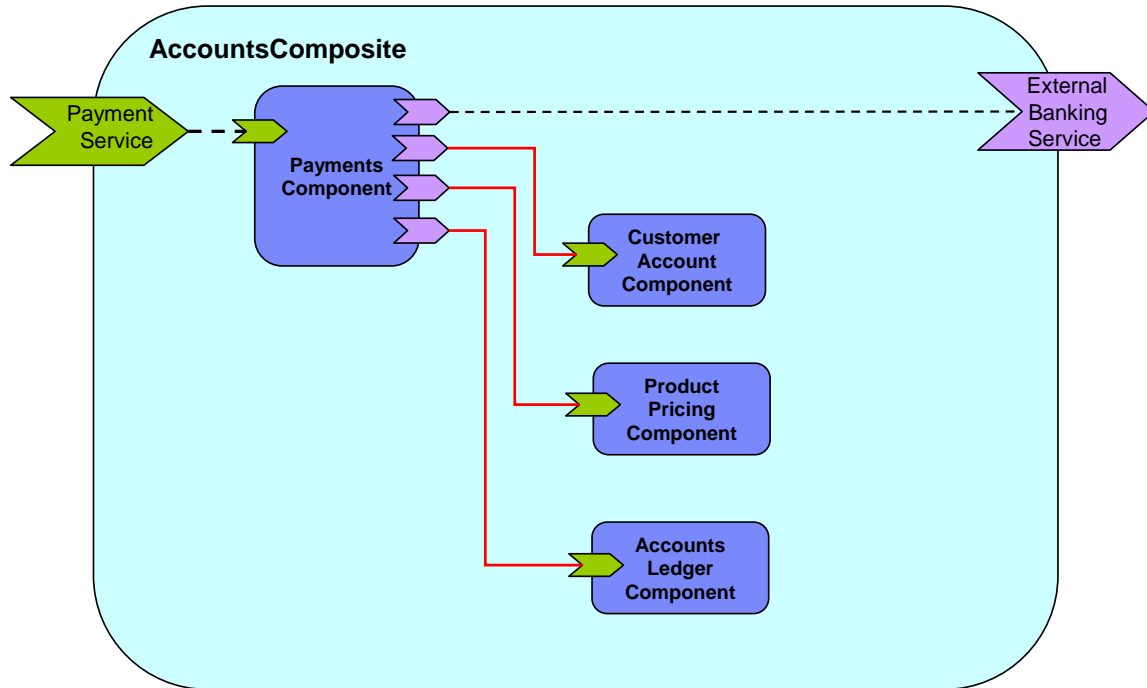


Figure 12: Example Composite for Autowire

First, the composite using explicit wires:

```

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

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

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

```

```

1788 <component name="CustomerAccountComponent" >
1789     <implementation.java class="com.foo.accounts.CustomerAccount" />
1790 </component>
1791
1792 <component name="ProductPricingComponent" >
1793     <implementation.composite class="com.foo.accounts.ProductPricing" />
1794 </component>
1795
1796 <component name="AccountsLedgerComponent" >
1797     <implementation.composite class="com.foo.accounts.AccountsLedger" />
1798 </component>
1799
1800 <reference name="ExternalBankingService"
1801     promote="PaymentsComponent/ExternalBankingService" />
1802
1803 </composite>
1804

```

1805 Secondly, the composite using autowire:

```

1806 <?xml version="1.0" encoding="UTF-8"?>
1807 <!-- Autowire Example - With autowire -->
1808 <composite xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
1809     xmlns="http://www.osoa.org/xmlns/sca/1.0"
1810     targetNamespace="http://foo.com"
1811     name="AccountComposite">
1812
1813     <service name="PaymentService" promote="PaymentsComponent">
1814         <interface.java class="com.foo.PaymentServiceInterface" />
1815     </service>
1816
1817     <component name="PaymentsComponent" autowire="true">
1818         <implementation.java class="com.foo.accounts.Payments" />
1819         <service name="PaymentService" />
1820         <reference name="CustomerAccountService" />
1821         <reference name="ProductPricingService" />
1822         <reference name="AccountsLedgerService" />
1823         <reference name="ExternalBankingService" />
1824     </component>
1825
1826     <component name="CustomerAccountComponent" >
1827         <implementation.java class="com.foo.accounts.CustomerAccount" />
1828     </component>
1829
1830     <component name="ProductPricingComponent" >
1831         <implementation.composite class="com.foo.accounts.ProductPricing" />
1832     </component>
1833
1834     <component name="AccountsLedgerComponent" >
1835         <implementation.composite class="com.foo.accounts.AccountsLedger" />
1836     </component>
1837
1838     <reference name="ExternalBankingService"
1839         promote="PaymentsComponent/ExternalBankingService" />
1840
1841 </composite>

```

1842 In this second case, autowire is set on for the PaymentsComponent and there are no explicit
1843 wires for any of its references – the wires are created automatically through autowire.

1844 **Note:** In the second example, it would be possible to omit all of the service and reference
1845 elements from the PaymentsComponent. They are left in for clarity, but if they are omitted, the

1846 component service and references still exist, since they are provided by the implementation used
1847 by the component.

1848

1849 1.6.5 Using Composites as Component Implementations

1850 Composites may form *component implementations* in higher-level composites – in other
1851 words the higher-level composites can have components which are implemented by composites.

1852 When a composite is used as a component implementation, it defines a boundary of visibility.
1853 Components within the composite cannot be referenced directly by the using component. The
1854 using component can only connect wires to the services and references of the used composite
1855 and set values for any properties of the composite. The internal construction of the composite is
1856 invisible to the using component.

1857 A composite used as a component implementation must also honor a *completeness contract*.
1858 The services, references and properties of the composite form a contract which is relied upon by
1859 the using component. The concept of completeness of the composite implies:

- 1860 • the composite must have at least one service or at least one reference.
1861 A component with no services and no references is not meaningful in terms of SCA, since
1862 it cannot be wired to anything – it neither provides nor consumes any services
1863
- 1864 • each service offered by the composite must be wired to a service of a component or to a
1865 composite reference.
1866 If services are left unwired, the implication is that some exception will occur at runtime if
1867 the service is invoked.

1868 The component type of a composite is defined by the set of service elements, reference elements
1869 and property elements that are the children of the composite element.

1870 Composites are used as component implementations through the use of the
1871 *implementation.composite* element as a child element of the component. The schema snippet
1872 for the implementation.composite element is:

1873

```
1874 <?xml version="1.0" encoding="ASCII"?>
1875 <!-- Composite Implementation schema snippet -->
1876 <composite xmlns="http://www.oesa.org/xmlns/sca/1.0"
1877     targetNamespace="xs:anyURI"
1878     name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?
1879     constrainingType="QName"?
1880     requires="list of xs:QName"? policySets="list of xs:QName"?>
1881     ...
1882
1883
1884     <component name="xs:NCName" autowire="xs:boolean"?
1885         requires="list of xs:QName"? policySets="list of xs:QName"?>*
1886         <implementation.composite name="xs:QName"/>?
1887         <service name="xs:NCName" requires="list of xs:QName"?
1888             policySets="list of xs:QName"?>*
1889             <interface/>?
1890             <binding uri="xs:anyURI" name="xs:QName"?
1891                 requires="list of xs:QName"
1892                 policySets="list of xs:QName"?/>*
1893             <callback?
1894                 <binding uri="xs:anyURI"? name="xs:QName"?
1895                     requires="list of xs:QName"?
1896                     policySets="list of xs:QName"?/>+
1897             </callback>
1898         </service>
```

```

1899     <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
1900         source="xs:string"? file="xs:anyURI"?>*
1901         property-value
1902     </property>
1903     <reference name="xs:NCName" target="list of xs:anyURI"?
1904         autowire="xs:boolean"? wiredByImpl="xs:boolean"?
1905         requires="list of xs:QName"? policySets="list of xs:QName"?
1906         multiplicity="0..1 or 1..1 or 0..n or 1..n"?/>*
1907     <interface/>?
1908     <binding uri="xs:anyURI"? name="xs:QName"?
1909         requires="list of xs:QName" policySets="list of xs:QName"?/>*
1910     <callback?>
1911         <binding uri="xs:anyURI"? name="xs:QName"?
1912             requires="list of xs:QName"?
1913             policySets="list of xs:QName"?/>+
1914     </callback>
1915 </reference>
1916 </component>
1917
1918     ...
1919
1920 </composite>
1921
1922

```

The implementation.composite element has the following attribute:

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

1.6.5.1 Example of Composite used as a Component Implementation

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

```

1931 <?xml version="1.0" encoding="UTF-8"?>
1932 <!-- CompositeComponent example -->
1933 <composite xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
1934     xsd:schemaLocation="http://www.osea.org/xmlns/sca/1.0
1935     file:/C:/Strategy/SCA/v09_oseaschemas/schemas/sca.xsd"
1936     xmlns="http://www.osea.org/xmlns/sca/1.0"
1937     targetNamespace="http://foo.com"
1938     xmlns:foo="http://foo.com"
1939     name="AccountComposite">
1940
1941     <service name="AccountService" promote="AccountServiceComponent">
1942         <interface.java interface="services.account.AccountService"/>
1943         <binding.ws port="AccountService#
1944             wsdl.endpoint(AccountService/AccountServiceSOAP)"/>
1945     </service>
1946
1947     <reference name="stockQuoteService"
1948         promote="AccountServiceComponent/StockQuoteService">
1949         <interface.java interface="services.stockquote.StockQuoteService"/>
1950         <binding.ws port="http://www.quickstockquote.com/StockQuoteService#
1951             wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
1952     </reference>
1953
1954     <property name="currency" type="xsd:string">EURO</property>
1955

```

```

1956     <component name="AccountServiceComponent">
1957         <implementation.composite name="foo:AccountServiceComposite1"/>
1958
1959         <reference name="AccountDataService" target="AccountDataService"/>
1960         <reference name="StockQuoteService"/>
1961
1962         <property name="currency" source="$currency"/>
1963     </component>
1964
1965     <component name="AccountDataService">
1966         <implementation.composite name="foo:AccountDataServiceComposite"/>
1967
1968         <property name="currency" source="$currency"/>
1969     </component>
1970
1971 </composite>
1972

```

1.6.6 Using Composites through Inclusion

1974 In order to assist team development, composites may be developed in the form of multiple
1975 physical artifacts that are merged into a single logical unit.

1976 A composite is defined in an *xxx.composite* file and the composite may receive additional
1977 content through the *inclusion of other composite* files.

1978 The semantics of included composites are that the content of the included composite is inlined
1979 into the using composite *xxx.composite* file through *include* elements in the using composite.
1980 The effect is one of *textual inclusion* – that is, the text content of the included composite is
1981 placed into the using composite in place of the include statement. The included composite
1982 element itself is discarded in this process – only its contents are included.

1983 The composite file used for inclusion can have any contents, but always contains a single
1984 *composite* element. The composite element may contain any of the elements which are valid as
1985 child elements of a composite element, namely components, services, references, wires and
1986 includes. There is no need for the content of an included composite to be complete, so that
1987 artifacts defined within the using composite or in another associated included composite file may
1988 be referenced. For example, it is permissible to have two components in one composite file while
1989 a wire specifying one component as the source and the other as the target can be defined in a
1990 second included composite file.

1991 It is an error if the (using) composite resulting from the inclusion is invalid – for example, if
1992 there are duplicated elements in the using composite (eg. two services with the same uri
1993 contributed by different included composites), or if there are wires with non-existent source or
1994 target.

1995 The following snippet shows the partial schema for the include element.

```

1996
1997 <?xml version="1.0" encoding="UTF-8"?>
1998 <!-- Include snippet -->
1999 <composite      xmlns="http://www.oesa.org/xmlns/sca/1.0"
2000                targetNamespace="xs:anyURI"
2001                name="xs:NCName" local="xs:boolean"? autowire="xs:boolean"?
2002                constrainingType="QName"?
2003                requires="list of xs:QName"? policySets="list of xs:QName"?>
2004
2005     ...
2006
2007     <include name="xs:QName"/>*
2008
2009     ...

```

2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028

2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040

```
</composite>
```

The include element has the following *attribute*:

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

1.6.6.1 Included Composite Examples

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

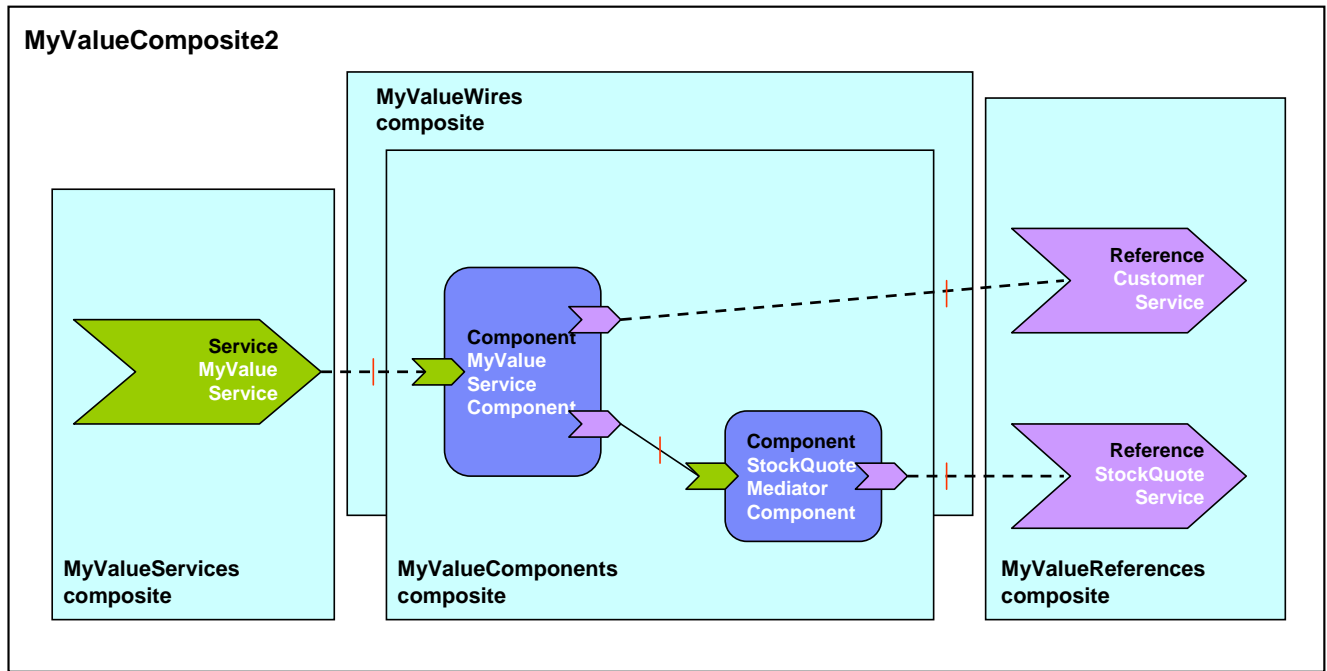


Figure 13 MyValueComposite2 built from 4 included composites

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

```
<?xml version="1.0" encoding="ASCII"?>  
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"  
targetNamespace="http://foo.com"
```

```

2041         xmlns:foo="http://foo.com"
2042         name="MyValueComposite2" >
2043
2044     <include name="foo:MyValueServices"/>
2045     <include name="foo:MyValueComponents"/>
2046     <include name="foo:MyValueReferences"/>
2047     <include name="foo:MyValueWires"/>
2048
2049 </composite>

```

2051 The following snippet shows the content of the MyValueServices.composite file.

```

2052
2053 <?xml version="1.0" encoding="ASCII"?>
2054 <composite xmlns="http://www.oesa.org/xmlns/sca/1.0"
2055           targetNamespace="http://foo.com"
2056           xmlns:foo="http://foo.com"
2057           name="MyValueServices" >
2058
2059     <service name="MyValueService" promote="MyValueServiceComponent">
2060       <interface.java interface="services.myvalue.MyValueService"/>
2061       <binding.ws port="http://www.myvalue.org/MyValueService#
2062                 wsdl.endpoint(MyValueService/MyValueServiceSOAP)"/>
2063     </service>
2064
2065 </composite>
2066

```

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

```

2068
2069 <?xml version="1.0" encoding="ASCII"?>
2070 <composite xmlns="http://www.oesa.org/xmlns/sca/1.0"
2071           targetNamespace="http://foo.com"
2072           xmlns:foo="http://foo.com"
2073           name="MyValueComponents" >
2074
2075     <component name="MyValueServiceComponent">
2076       <implementation.java class="services.myvalue.MyValueServiceImpl"/>
2077       <property name="currency">EURO</property>
2078     </component>
2079
2080     <component name="StockQuoteMediatorComponent">
2081       <implementation.java class="services.myvalue.SQMediatorImpl"/>
2082       <property name="currency">EURO</property>
2083     </component>
2084
2085 </composite>
2086

```

2087 The following snippet shows the content of the MyValueReferences.composite file.

```

2088
2089 <?xml version="1.0" encoding="ASCII"?>
2090 <composite xmlns="http://www.oesa.org/xmlns/sca/1.0"
2091           targetNamespace="http://foo.com"
2092           xmlns:foo="http://foo.com"
2093           name="MyValueReferences" >
2094
2095     <reference name="CustomerService"

```

```

2096         promote="MyValueServiceComponent/CustomerService">
2097         <interface.java interface="services.customer.CustomerService"/>
2098         <binding.sca/>
2099     </reference>
2100
2101     <reference name="StockQuoteService" promote="StockQuoteMediatorComponent">
2102         <interface.java interface="services.stockquote.StockQuoteService"/>
2103         <binding.ws port="http://www.stockquote.org/StockQuoteService#"
2104             wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
2105     </reference>
2106
2107 </composite>

```

2108 The following snippet shows the content of the MyValueWires.composite file.

```

2109
2110 <?xml version="1.0" encoding="ASCII"?>
2111 <composite xmlns="http://www.oesa.org/xmlns/sca/1.0"
2112     targetNamespace="http://foo.com"
2113     xmlns:foo="http://foo.com"
2114     name="MyValueWires" >
2115
2116     <wire source="MyValueServiceComponent/stockQuoteService"
2117         target="StockQuoteMediatorComponent"/>
2118
2119 </composite>

```

2120 1.6.7 Composites which Include Component Implementations of Multiple Types

2121

2122 A Composite containing multiple components MAY have multiple component implementation
2123 types. For example, a Composite may include one component with a Java POJO as its
2124 implementation and another component with a BPEL process as its implementation.

2125

2126 1.6.8 ConstrainingType

2127 SCA allows a component, and its associated implementation, to be constrained by a
2128 **constrainingType**. The constrainingType element provides assistance in developing top-down
2129 usecases in SCA, where an architect or assembler can define the structure of a composite,
2130 including the required form of component implementations, before any of the implementations
2131 are developed.

2132 A constrainingType is expressed as an element which has services, reference and properties as
2133 child elements and which can have intents applied to it. The constrainingType is independent of
2134 any implementation. Since it is independent of an implementation it cannot contain any
2135 implementation-specific configuration information or defaults. Specifically, it cannot contain
2136 bindings, policySets, property values or default wiring information. The constrainingType is
2137 applied to a component through a constrainingType attribute on the component.

2138 A constrainingType provides the "shape" for a component and its implementation. Any
2139 component configuration that points to a constrainingType is constrained by this shape. The
2140 constrainingType specifies the services, references and properties that must be implemented.
2141 This provides the ability for the implementer to program to a specific set of services, references
2142 and properties as defined by the constrainingType. Components are therefore configured
2143 instances of implementations and are constrained by an associated constrainingType.

2144 If the configuration of the component or its implementation do not conform to the
2145 constrainingType, it is an error.

2146 A constrainingType is represented by a **constrainingType** element. The following snippet
2147 shows the pseudo-schema for the composite element.

2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172

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

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

    <reference name="xs:NCName"
        multiplicity="0..1 or 1..1 or 0..n or 1..n"?
        requires="list of xs:QName"?>*
        <interface/?>
    </reference>

    <property name="xs:NCName" (type="xs:QName" | element="xs:QName")
        many="xs:boolean"? mustSupply="xs:boolean"?>*
        default-property-value?
    </property>
</constrainingType>
```

2173 The constrainingType element has the following **attributes**:

- 2174 • **name (required)** – the name of the constrainingType. The form of a constrainingType
2175 name is an XML QName, in the namespace identified by the targetNamespace attribute.
- 2176 • **targetNamespace (optional)** – an identifier for a target namespace into which the
2177 constrainingType is declared
- 2178 • **requires (optional)** – a list of policy intents. See [the Policy Framework specification](#)
2179 [\[10\]](#) for a description of this attribute.

2180 ConstrainingType contains **zero or more properties, services, references**.

2181
2182
2183
2184
2185
2186
2187

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

2188
2189
2190
2191
2192
2193
2194
2195

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

2196
2197
2198
2199
2200

The constrainingType can include required intents on any element. Those intents are applied to any component that uses that constrainingType. In other words, if requires="reliability" exists on a constrainingType, or its child service or reference elements, then a constrained component or its implementation must include requires="reliability" on the component or implementation or on its corresponding service or reference. Note that the component or implementation may use

2201 a qualified form of an intent specified in unqualified form in the constrainingType, but if the
2202 constrainingType uses the qualified form, then the component or implementation must also use
2203 the qualified form, otherwise there is an error.

2204 A constrainingType can be applied to an implementation. In this case, the implementation's
2205 componentType has a constrainingType attribute set to the QName of the constrainingType.

2206

2207 **1.6.8.1 Example constrainingType**

2208

2209 The following snippet shows the contents of the component called "MyValueServiceComponent"
2210 which is constrained by the constrainingType myns:CT. The componentType associated with the
2211 implementation is also shown.

2212

```
2213 <component name="MyValueServiceComponent" constrainingType="myns:CT">
2214   <implementation.java class="services.myvalue.MyValueServiceImpl"/>
2215   <property name="currency">EURO</property>
2216   <reference name="customerService" target="CustomerService">
2217     <binding.ws ...>
2218   <reference name="StockQuoteService"
2219     target="StockQuoteMediatorComponent"/>
2220 </component>

2221
2222 <constrainingType name="CT"
2223   targetNamespace="http://myns.com">
2224   <service name="MyValueService">
2225     <interface.java interface="services.myvalue.MyValueService"/>
2226   </service>
2227   <reference name="customerService">
2228     <interface.java interface="services.customer.CustomerService"/>
2229   </reference>
2230   <reference name="stockQuoteService">
2231     <interface.java interface="services.stockquote.StockQuoteService"/>
2232   </reference>
2233   <property name="currency" type="xsd:string"/>
2234 </constrainingType>
```

2235 The component MyValueServiceComponent is constrained by the constrainingType CT which
2236 means that it must provide:

- 2237 • service **MyValueService** with the interface services.myvalue.MyValueService
- 2238 • reference **customerService** with the interface services.stockquote.StockQuoteService
- 2239 • reference **stockQuoteService** with the interface services.stockquote.StockQuoteService
- 2240 • property **currency** of type xsd:string.

1.7 Binding

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

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

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

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

2296
2297 `</composite>`
2298

2299 The element name of the binding element is architected; it is in itself a qualified name. The first
2300 qualifier is always named “binding”, and the second qualifier names the respective binding-type
2301 (e.g. binding.composite, binding.ws, binding.ejb, binding.eis).

2302
2303 A binding element has the following attributes:

- 2304 • **uri (optional)** - has the following semantic.
 - 2305 ○ For a binding of a **reference** the URI attribute defines the target URI of the
2306 reference (either the component/service for a wire to an endpoint within the SCA
2307 domain or the accessible address of some endpoint outside the SCA domain). It is
2308 optional for references defined in composites used as component implementations,
2309 but required for references defined in composites contributed to SCA domains. The
2310 URI attribute of a reference of a composite can be reconfigured by a component in
2311 a containing composite using the composite as an implementation. Some binding
2312 types may require that the address of the target service uses more than a simple
2313 URI (such as a WS-Addressing endpoint reference). In those cases, the binding
2314 type will define the additional attributes or sub-elements that are necessary to
2315 identify the service.
 - 2316 ○ For a binding of a **service** the URI attribute defines the URI relative to the
2317 component which contributes the service to the SCA domain. The default value for
2318 the URI is the the value of the name attribute of the binding.
- 2319 • **name (optional)** – a name for the binding instance (a QName). The name attribute
2320 allows distinction between multiple binding elements on a single service or reference. The
2321 default value of the name attribute is the service or reference name. When a service or
2322 reference has multiple bindings, only one can have the default value; all others must have
2323 a value specified that is unique within the service or reference. The name also permits the
2324 binding instance to be referenced from elsewhere – particularly useful for some types of
2325 binding, which can be declared in a definitions document as a template and referenced
2326 from other binding instances, simplifying the definition of more complex binding instances
2327 (see [the JMS Binding specification \[11\]](#) for examples of this referencing).
- 2328 • **requires (optional)** - a list of policy intents. See the [Policy Framework specification \[10\]](#)
2329 for a description of this attribute.
- 2330 • **policySets (optional)** – a list of policy sets. See the [Policy Framework specification \[10\]](#)
2331 for a description of this attribute.

2332 When multiple bindings exist for an service, it means that the service is available by any of the
2333 specified bindings. The technique that the SCA runtime uses to choose among available bindings
2334 is left to the implementation and it may include additional (nonstandard) configuration.
2335 Whatever technique is used SHOULD be documented.

2336 Services and References can always have their bindings overridden at the SCA domain level,
2337 unless restricted by Intents applied to them.

2338 The following sections describe the SCA and Web service binding type in detail.

2339 2340 **1.7.1 Messages containing Data not defined in the Service Interface** 2341

2342 It is possible for a message to include information that is not defined in the interface used to
2343 define the service, for instance information may be contained in SOAP headers or as MIME
2344 attachments.

2345 Implementation types MAY make this information available to component implementations in
2346 their execution context. These implementation types must indicate how this information is
2347 accessed and in what form they are presented.

2348

2349 **1.7.2 Form of the URI of a Deployed Binding**

2350

2351 **1.7.2.1 Constructing Hierarchical URIs**

2352 Bindings that use hierarchical URI schemes construct the effective URI with a combination of the
2353 following pieces:

2354 Base System URI for a scheme / Component URI / Service Binding URI

2355

2356 Each of these components deserves addition definition:

2357 **Base Domain URI for a scheme.** An SCA domain should define a base URI for each
2358 hierarchical URI scheme on which it intends to provide services.

2359 For example: the HTTP and HTTPS schemes would each have their own base URI defined for the
2360 domain. An example of a scheme that is not hierarchical, and therefore will have no base URI is
2361 the "jms:" scheme.

2362 **Component URI.** The component URI above is for a component that is deployed in the SCA
2363 Domain. The URI of a component defaults to the name of the component, which is used as a
2364 relative URI. The component may have a specified URI value. The specified URI value may be
2365 an absolute URI in which case it becomes the Base URI for all the services belonging to the
2366 component. If the specified URI value is a relative URI, it is used as the Component URI value
2367 above.

2368 **Service Binding URI.** The Service Binding URI is the relative URI specified in the "uri" attribute
2369 of a binding element of the service. The default value of the attribute is value of the binding's
2370 name attribute treated as a relative URI. If multiple bindings for a single service use the same
2371 scheme (e.g. HTTP), then only one of the bindings may depend on the default value for the uri
2372 attribute, i.e. only one may use the default binding name. The service binding URI may also be
2373 absolute, in which case the absolute URI fully specifies the full URI of the service. Some
2374 deployment environments may not support the use of absolute URIs in service bindings.

2375 Where a component has only a single service, the default value of the Service Binding URI is null,
2376 so that the effective URI is:

2377 Base Domain URI for a scheme / Component URI

2378 This shortened form of the URI is consistent with the shortened form for the wire target URI used
2379 when wiring references to services

2380 Services deployed into the Domain (as opposed to services of components) have a URI that does
2381 not include a component name, i.e.:

2382 Base Domain URI for a scheme / Service Binding URI

2383 The name of the containing composite does not contribute to the URI of any service.

2384 For example, a service where the Base URI is "http://acme.com", the component is named
2385 "stocksComponent" and the service binding name is "getQuote", the URI would look like this:

2386 http://acme.com/stocksComponent/getQuote

2387 Allowing a binding's relative URI to be specified that differs from the name of the service allows
2388 the URI hierarchy of services to be designed independently of the organization of the domain.

2389 It is good practice to design the URI hierarchy to be independent of the domain organization, but
2390 there may be times when domains are initially created using the default URI hierarchy. When

2391 this is the case, the organization of the domain can be changed, while maintaining the form of
2392 the URI hierarchy, by giving appropriate values to the *uri* attribute of select elements. Here is
2393 an example of a change that can be made to the organization while maintaining the existing
2394 URIs:

2395 To move a subset of the services out of one component (say "foo") to a new component (say
2396 "bar"), the new component should have bindings for the moved services specify a URI
2397 "../foo/MovedService"..

2398 The URI attribute may also be used in order to create shorter URIs for some endpoints, where
2399 the component name may not be present in the URI at all. For example, if a binding has a *uri*
2400 attribute of "../myService" the component name will not be present in the URI.

2401 **1.7.2.2 Non-hierarchical URIs**

2402 Bindings that use non-hierarchical URI schemes (such as jms: or mailto:) may optionally make
2403 use of the "uri" attribute, which is the complete representation of the URI for that service
2404 binding. Where the binding does not use the "uri" attribute, the binding must offer a different
2405 mechanism for specifying the service address.

2406 **1.7.2.3 Determining the URI scheme of a deployed binding**

2407 One of the things that needs to be determined when building the effective URI of a deployed
2408 binding (i.e. endpoint) is the URI scheme. The process of determining the endpoint URI scheme
2409 is binding type specific.

2410 If the binding type supports a single protocol then there is only one URI scheme associated with
2411 it. In this case, that URI scheme is used.

2412 If the binding type supports multiple protocols, the binding type implementation determines the
2413 URI scheme by introspecting the binding configuration, which may include the policy sets
2414 associated with the binding.

2415 A good example of a binding type that supports multiple protocols is binding.ws, which can be
2416 configured by referencing either an "abstract" WSDL element (i.e. portType or interface) or a
2417 "concrete" WSDL element (i.e. binding, port or endpoint). When the binding references a
2418 PortType or Interface, the protocol and therefore the URI scheme is derived from the
2419 intents/policy sets attached to the binding. When the binding references a "concrete" WSDL
2420 element, there are two cases:

- 2421 1) The referenced WSDL binding element uniquely identifies a URI scheme. This is the most
2422 common case. In this case, the URI scheme is given by the protocol/transport specified in the
2423 WSDL binding element.
- 2424 2) The referenced WSDL binding element doesn't uniquely identify a URI scheme. For example,
2425 when HTTP is specified in the @transport attribute of the SOAP binding element, both "http"
2426 and "https" could be used as valid URI schemes. In this case, the URI scheme is determined
2427 by looking at the policy sets attached to the binding.

2428 It's worth noting that an intent supported by a binding type may completely change the behavior
2429 of the binding. For example, when the intent "confidentiality/transport" is required by an HTTP
2430 binding, SSL is turned on. This basically changes the URI scheme of the binding from "http" to
2431 "https".

2432

2433 **1.7.3 SCA Binding**

2434 The SCA binding element is defined by the following schema.

2435

2436 `<binding.sca />`

2437

2438 The SCA binding can be used for service interactions between references and services contained
2439 within the SCA domain. The way in which this binding type is implemented is not defined by the
2440 SCA specification and it can be implemented in different ways by different SCA runtimes. The
2441 only requirement is that the required qualities of service must be implemented for the SCA
2442 binding type. The SCA binding type is *not* intended to be an interoperable binding type. For
2443 interoperability, an interoperable binding type such as the Web service binding should be used.

2444 A service or reference definition with no binding element specified uses the SCA binding.
2445 <binding.sca/> would only have to be specified in override cases, or when you specify a set of
2446 bindings on a service or reference definition and the SCA binding should be one of them.

2447

2448 If the interface of the service or reference is local, then the local variant of the SCA binding will
2449 be used. If the interface of the service or reference is remotable, then either the local or remote
2450 variant of the SCA binding will be used depending on whether source and target are co-located
2451 or not.

2452 If a reference specifies an URI via its uri attribute, then this provides the default wire to a service
2453 provided by another domain level component. The value of the URI has to be as follows:

- 2454 • <domain-component-name>/<service-name>

2455

2456 1.7.3.1 Example SCA Binding

2457 The following snippet shows the MyValueComposite.composite file for the MyValueComposite
2458 containing the service element for the MyValueService and a reference element for the
2459 StockQuoteService. Both the service and the reference use an SCA binding. The target for the
2460 reference is left undefined in this binding and would have to be supplied by the composite in
2461 which this composite is used.

2462

```
2463 <?xml version="1.0" encoding="ASCII"?>
2464 <!-- Binding SCA example -->
2465 <composite xmlns="http://www.oesa.org/xmlns/sca/1.0"
2466           targetNamespace="http://foo.com"
2467           name="MyValueComposite" >
2468
2469     <service name="MyValueService" promote="MyValueComponent">
2470       <interface.java interface="services.myvalue.MyValueService"/>
2471       <binding.sca/>
2472       ...
2473     </service>
2474
2475     ...
2476
2477     <reference name="StockQuoteService"
2478 promote="MyValueComponent/StockQuoteReference">
2479       <interface.java interface="services.stockquote.StockQuoteService"/>
2480       <binding.sca/>
2481     </reference>
2482
2483 </composite>
```

2484

2485 1.7.4 Web Service Binding

2486 SCA defines a Web services binding. This is described in [a separate specification document \[9\]](#).

2487

2488 **1.7.5 JMS Binding**

2489 SCA defines a JMS binding. This is described in [a separate specification document \[11\]](#).

2490 1.8 SCA Definitions

2491 There are a variety of SCA artifacts which are generally useful and which are not specific to a
2492 particular composite or a particular component. These shared artifacts include intents, policy
2493 sets, bindings, binding type definitions and implementation type definitions.

2494 All of these artifacts within an SCA Domain are defined in a global, SCA Domain-wide file named
2495 definitions.xml. The definitions.xml file contains a definitions element that conforms to the
2496 following pseudo-schema snippet:

```
2497 <?xml version="1.0" encoding="ASCII"?>  
2498 <!-- Composite schema snippet -->  
2499 <definitions xmlns="http://www.osoa.org/xmlns/sca/1.0"  
2500             targetNamespace="xs:anyURI">  
2501  
2502     <sca:intent/*>  
2503  
2504     <sca:policySet/*>  
2505  
2506     <sca:binding/*>  
2507  
2508     <sca:bindingType/*>  
2509  
2510     <sca:implementationType/*>  
2511  
2512 </definitions>
```

2513 The definitions element has the following attribute:

- 2514 • **targetNamespace (required)** – the namespace into which the child elements of this
2515 definitions element are placed (used for artifact resolution)

2516 The definitions element contains optional child elements – intent, policySet, binding, bindingtype
2517 and implementationType. These elements are described elsewhere in this specification or in [the](#)
2518 [SCA Policy Framework specification \[10\]](#). The use of the elements declared within a definitions
2519 element is described in the SCA Policy Framework specification [10] and in [the JMS Binding](#)
2520 [specification \[11\]](#).

1.9 Extension Model

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

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

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

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

1.9.1 Defining an Interface Type

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

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2006 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.oesa.org/xmlns/sca/1.0"
  xmlns:sca="http://www.oesa.org/xmlns/sca/1.0"
  elementFormDefault="qualified">
  ...
  <element name="interface" type="sca:Interface" abstract="true"/>
  <complexType name="Interface"/>
  ...
</schema>
```

In the following snippet we show 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://www.oesa.org/xmlns/sca/1.0"
  xmlns:sca="http://www.oesa.org/xmlns/sca/1.0">
```



```

2573
2574     <element name="interface.java" type="sca:JavaInterface"
2575         substitutionGroup="sca:interface"/>
2576     <complexType name="JavaInterface">
2577         <complexContent>
2578             <extension base="sca:Interface">
2579                 <attribute name="interface" type="NCName" use="required"/>
2580             </extension>
2581         </complexContent>
2582     </complexType>
2583 </schema>

```

2584 In the following snippet we show an example of how the base definition can be extended by
2585 other specifications to support a new interface not defined in the SCA specifications. The snippet
2586 shows the definition of the **my-interface-extension** element and the **my-interface-**
2587 **extension-type** type.

```

2588 <?xml version="1.0" encoding="UTF-8"?>
2589 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2590     targetNamespace="http://www.example.org/myextension"
2591     xmlns:sca="http://www.oesa.org/xmlns/sca/1.0"
2592     xmlns:tns="http://www.example.org/myextension">
2593
2594     <element name="my-interface-extension" type="tns:my-interface-extension-type"
2595         substitutionGroup="sca:interface"/>
2596     <complexType name="my-interface-extension-type">
2597         <complexContent>
2598             <extension base="sca:Interface">
2599                 ...
2600             </extension>
2601         </complexContent>
2602     </complexType>
2603 </schema>
2604

```

2605 1.9.2 Defining an Implementation Type

2606 The following snippet shows the base definition for the **implementation** element and
2607 **Implementation** type contained in **sca-core.xsd**; see appendix for complete schema.

```

2608
2609 <?xml version="1.0" encoding="UTF-8"?>
2610 <!-- (c) Copyright SCA Collaboration 2006 -->
2611 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2612     targetNamespace="http://www.oesa.org/xmlns/sca/1.0"
2613     xmlns:sca="http://www.oesa.org/xmlns/sca/1.0"
2614     elementFormDefault="qualified">
2615
2616     ...
2617
2618     <element name="implementation" type="sca:Implementation" abstract="true"/>
2619     <complexType name="Implementation"/>
2620
2621     ...
2622
2623 </schema>
2624

```

2625 In the following snippet we show how the base definition is extended to support Java
2626 implementation. The snippet shows the definition of the **implementation.java** element and the
2627 **JavaImplementation** type contained in **sca-implementation-java.xsd**.

2628

```

2629 <?xml version="1.0" encoding="UTF-8"?>
2630 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2631         targetNamespace="http://www.osea.org/xmlns/sca/1.0"
2632         xmlns:sca="http://www.osea.org/xmlns/sca/1.0">
2633
2634     <element name="implementation.java" type="sca:JavaImplementation"
2635             substitutionGroup="sca:implementation"/>
2636     <complexType name="JavaImplementation">
2637         <complexContent>
2638             <extension base="sca:Implementation">
2639                 <attribute name="class" type="NCName" use="required"/>
2640             </extension>
2641         </complexContent>
2642     </complexType>
2643 </schema>

```

2644 In the following snippet we show an example of how the base definition can be extended by
2645 other specifications to support a new implementation type not defined in the SCA specifications.
2646 The snippet shows the definition of the *my-impl-extension* element and the *my-impl-*
2647 *extension-type* type.

```

2648 <?xml version="1.0" encoding="UTF-8"?>
2649 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2650         targetNamespace="http://www.example.org/myextension"
2651         xmlns:sca="http://www.osea.org/xmlns/sca/1.0"
2652         xmlns:tns="http://www.example.org/myextension">
2653
2654     <element name="my-impl-extension" type="tns:my-impl-extension-type"
2655             substitutionGroup="sca:implementation"/>
2656     <complexType name="my-impl-extension-type">
2657         <complexContent>
2658             <extension base="sca:Implementation">
2659                 ...
2660             </extension>
2661         </complexContent>
2662     </complexType>
2663 </schema>
2664

```

2665 In addition to the definition for the new implementation instance element, there needs to be an
2666 associated implementationType element which provides metadata about the new implementation
2667 type. The pseudo schema for the implementationType element is shown in the following snippet:

```

2668 <implementationType type="xs:QName"
2669                 alwaysProvides="list of intent xs:QName"
2670                 mayProvide="list of intent xs:QName"/>
2671

```

2672 The implementation type has the following attributes:

- 2673 • **type (required)** – the type of the implementation to which this implementationType
2674 element applies. This is intended to be the QName of the implementation element for
2675 the implementation type, such as "sca:implementation.java"
- 2676 • **alwaysProvides (optional)** – a set of intents which the implementation type always
2677 provides. See [the Policy Framework specification \[10\]](#) for details.
- 2678 • **mayProvide (optional)** – a set of intents which the implementation type may provide.
2679 See [the Policy Framework specification \[10\]](#) for details.

2680

2681 1.9.3 Defining a Binding Type

2682 The following snippet shows the base definition for the *binding* element and *Binding* type
2683 contained in *sca-core.xsd*; see appendix for complete schema.

2684

```
2685 <?xml version="1.0" encoding="UTF-8"?>
2686 <!-- binding type schema snippet -->
2687 <!-- (c) Copyright SCA Collaboration 2006, 2007 -->
2688 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2689         targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
2690         xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
2691         elementFormDefault="qualified">
2692
2693     ...
2694
2695     <element name="binding" type="sca:Binding" abstract="true"/>
2696     <complexType name="Binding">
2697         <attribute name="uri" type="anyURI" use="optional"/>
2698         <attribute name="name" type="NCName" use="optional"/>
2699         <attribute name="requires" type="sca:listOfQNames" use="optional"/>
2700         <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
2701     </complexType>
2702
2703     ...
2704
2705 </schema>
```

2706 In the following snippet we show how the base definition is extended to support Web service
2707 binding. The snippet shows the definition of the *binding.ws* element and the
2708 *WebServiceBinding* type contained in *sca-binding-webservice.xsd*.

2709

```
2710 <?xml version="1.0" encoding="UTF-8"?>
2711 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2712         targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
2713         xmlns:sca="http://www.osoa.org/xmlns/sca/1.0">
2714
2715     <element name="binding.ws" type="sca:WebServiceBinding"
2716             substitutionGroup="sca:binding"/>
2717     <complexType name="WebServiceBinding">
2718         <complexContent>
2719             <extension base="sca:Binding">
2720                 <attribute name="port" type="anyURI" use="required"/>
2721             </extension>
2722         </complexContent>
2723     </complexType>
2724 </schema>
```

2725 In the following snippet we show an example of how the base definition can be extended by
2726 other specifications to support a new binding not defined in the SCA specifications. The snippet
2727 shows the definition of the *my-binding-extension* element and the *my-binding-extension-*
2728 *type* type.

```
2729 <?xml version="1.0" encoding="UTF-8"?>
2730 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2731         targetNamespace="http://www.example.org/myextension"
2732         xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
2733         xmlns:tns="http://www.example.org/myextension">
2734
2735     <element name="my-binding-extension" type="tns:my-binding-extension-type"
```

```

2736         substitutionGroup="sca:binding"/>
2737     <complexType name="my-binding-extension-type">
2738         <complexContent>
2739             <extension base="sca:Binding">
2740                 ...
2741             </extension>
2742         </complexContent>
2743     </complexType>
2744 </schema>
2745

```

2746 In addition to the definition for the new binding instance element, there needs to be an
2747 associated bindingType element which provides metadata about the new binding type. The
2748 pseudo schema for the bindingType element is shown in the following snippet:

```

2749 <bindingType type="xs:QName"
2750     alwaysProvides="list of intent QNames"?
2751     mayProvide = "list of intent QNames"?/>
2752

```

2753 The binding type has the following attributes:

- 2754 • **type (required)** – the type of the binding to which this bindingType element applies.
2755 This is intended to be the QName of the binding element for the binding type, such as
2756 "sca:binding.ws"
- 2757 • **alwaysProvides (optional)** – a set of intents which the binding type always provides.
2758 See [the Policy Framework specification \[10\]](#) for details.
- 2759 • **mayProvide (optional)** – a set of intents which the binding type may provide. See [the](#)
2760 [Policy Framework specification \[10\]](#) for details.

2761

2762 **1.10 Packaging and Deployment**

2763 **1.10.1 Domains**

2764 An **SCA Domain** represents a complete runtime configuration, potentially distributed over a
2765 series of interconnected runtime nodes.

2766 A single SCA domain defines the boundary of visibility for all SCA mechanisms. For example,
2767 SCA wires can only be used to connect components within a single SCA domain. Connections to
2768 services outside the domain must use binding specific mechanisms for addressing services (such
2769 as WSDL endpoint URIs). Also, SCA mechanisms such as intents and policySets can only be
2770 used in the context of a single domain. In general, external clients of a service that is developed
2771 and deployed using SCA should not be able to tell that SCA was used to implement the service –
2772 it is an implementation detail.

2773 The size and configuration of an SCA Domain is not constrained by the SCA Assembly
2774 specification and is expected to be highly variable. An SCA Domain typically represents an area
2775 of business functionality controlled by a single organization. For example, an SCA Domain may
2776 be the whole of a business, or it may be a department within a business.

2777 As an example, for the accounts department in a business, the SCA Domain might cover all
2778 finance-related functions, and it might contain a series of composites dealing with specific areas
2779 of accounting, with one for Customer accounts and another dealing with Accounts Payable.

2780 An SCA domain has the following:

- 2781 • A virtual domain-level composite whose components are deployed and running
- 2782 • A set of *installed contributions* that contain implementations, interfaces and other artifacts
2783 necessary to execute components
- 2784 • A set of logical services for manipulating the set of contributions and the virtual domain-
2785 level composite.

2786 The information associated with an SCA domain can be stored in many ways, including but not
2787 limited to a specific filesystem structure or a repository.

2788 **1.10.2 Contributions**

2789 An SCA domain may require a large number of different artifacts in order to work. These
2790 artifacts include artifacts defined by SCA and other artifacts such as object code files and
2791 interface definition files. The SCA-defined artifact types are all XML documents. The root
2792 elements of the different SCA definition documents are: composite, componentType,
2793 constrainingType and definitions. XML artifacts that are not defined by SCA but which may be
2794 needed by an SCA domain include XML Schema documents, WSDL documents, and BPEL
2795 documents. SCA constructs, like other XML-defined constructs, use XML qualified names for
2796 their identity (i.e. namespace + local name).

2797 Non-XML artifacts are also required within an SCA domain. The most obvious examples of such
2798 non-XML artifacts are Java, C++ and other programming language files necessary for component
2799 implementations. Since SCA is extensible, other XML and non-XML artifacts may also be
2800 required.

2801 SCA defines an interoperable packaging format for contributions (ZIP), as specified below. This
2802 format is not the only packaging format that an SCA runtime can use. SCA allows many different
2803 packaging formats, but requires that the ZIP format be supported. When using the ZIP format for
2804 deploying a contribution, this specification does not specify whether that format is retained after
2805 deployment. For example, a Java EE based SCA runtime may convert the ZIP package to an EAR
2806 package. SCA expects certain characteristics of any packaging:

- 2807 • It must be possible to present the artifacts of the packaging to SCA as a hierarchy of
2808 resources based off of a single root
- 2809 • A directory resource should exist at the root of the hierarchy named META-INF

- 2810
- 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.
- 2811
- 2812

2813 The same document also optionally lists namespaces of constructs that are defined within
2814 the contribution and which may be used by other contributions

2815 Optionally, additional elements may exist that list the namespaces of constructs that are
2816 needed by the contribution and which must be found elsewhere, for example in other
2817 contributions. These optional elements may not be physically present in the packaging,
2818 but may be generated based on the definitions and references that are present, or they
2819 may not exist at all if there are no unresolved references.

2820
2821 See the section "SCA Contribution Metadata Document" for details of the format of this
2822 file.

2823 To illustrate that a variety of packaging formats can be used with SCA, the following are
2824 examples of formats that might be used to package SCA artifacts and metadata (as well as other
2825 artifacts) as a contribution:

- 2826
- A filesystem directory
- 2827
- An OSGi bundle
- 2828
- A compressed directory (zip, gzip, etc)
- 2829
- A JAR file (or its variants – WAR, EAR, etc)

2830 Contributions do not contain other contributions. If the packaging format is a JAR file that
2831 contains other JAR files (or any similar nesting of other technologies), the internal files are not
2832 treated as separate SCA contributions. It is up to the implementation to determine whether the
2833 internal JAR file should be represented as a single artifact in the contribution hierarchy or
2834 whether all of the contents should be represented as separate artifacts.

2835 A goal of SCA's approach to deployment is that the contents of a contribution should not need to
2836 be modified in order to install and use the contents of the contribution in a domain.

2837

2838 **1.10.2.1 SCA Artifact Resolution**

2839 Contributions may be self-contained, in that all of the artifacts necessary to run the contents of
2840 the contribution are found within the contribution itself. However, it may also be the case that
2841 the contents of the contribution make one or many references to artifacts that are not contained
2842 within the contribution. These references may be to SCA artifacts or they may be to other
2843 artifacts such as WSDL files, XSD files or to code artifacts such as Java class files and BPEL
2844 scripts.

2845 A contribution may use some artifact-related or packaging-related means to resolve artifact
2846 references. Examples of such mechanisms include:

- 2847
- wsdlLocation and schemaLocation attributes in references to WSDL and XSD schema
2848 artifacts respectively
- 2849
- OSGi bundle mechanisms for resolving Java class and related resource dependencies

2850 Where present, these mechanisms must be used to resolve artifact dependencies.

2851 SCA also provides an artifact resolution mechanism. The SCA artifact resolution mechanisms are
2852 used either where no other mechanisms are available, or in cases where the mechanisms used
2853 by the various contributions in the same SCA Domain are different. An example of the latter
2854 case is where an OSGi Bundle is used for one contribution but where a second contribution used
2855 by the first one is not implemented using OSGi - eg the second contribution is a mainframe
2856 COBOL service whose interfaces are declared using WSDL which must be accessed by the first
2857 contribution.

2858 The SCA artifact resolution is likely to be most useful for SCA domains containing heterogeneous
2859 mixtures of contribution, where artifact-related or packaging-related mechanisms are unlikely to
2860 work across different kinds of contribution.

2861 SCA artifact resolution works on the principle that a contribution which needs to use artifacts
2862 defined elsewhere expresses these dependencies using *import* statements in metadata
2863 belonging to the contribution. A contribution controls which artifacts it makes available to other
2864 contributions through *export* statements in metadata attached to the contribution.

2865

2866 1.10.2.2 SCA Contribution Metadata Document

2867 The contribution optionally contains a document that declares runnable composites, exported
2868 definitions and imported definitions. The document is found at the path of META-INF/sca-
2869 contribution.xml relative to the root of the contribution. Frequently some SCA metadata may
2870 need to be specified by hand while other metadata is generated by tools (such as the <import>
2871 elements described below). To accommodate this, it is also possible to have an identically
2872 structured document at META-INF/sca-contribution-generated.xml. If this document exists (or is
2873 generated on an as-needed basis), it will be merged into the contents of sca-contribution.xml,
2874 with the entries in sca-contribution.xml taking priority if there are any conflicting declarations.

2875 The format of the document is:

```
2876 <?xml version="1.0" encoding="ASCII"?>  
2877 <!-- sca-contribution pseudo-schema -->  
2878 <contribution xmlns=http://www.osoa.org/xmlns/sca/1.0>  
2879  
2880     <deployable composite="xs:QName"/>*  
2881     <import namespace="xs:String" location="xs:AnyURI"?/>*  
2882     <export namespace="xs:String"/>*  
2883  
2884 </contribution>
```

2885

2887 **deployable element:** Identifies a composite which is a composite within the contribution that is
2888 a composite intended for potential inclusion into the virtual domain-level composite. Other
2889 composites in the contribution are not intended for inclusion but only for use by other
2890 composites. New composites can be created for a contribution after it is installed, by using the
2891 [add Deployment Composite](#) capability and the add To Domain Level Composite capability.

- 2892 • **composite (required)** – The QName of a composite within the contribution.

2893

2894 **Export element:** A declaration that artifacts belonging to a particular namespace are exported
2895 and are available for use within other contributions. An export declaration in a contribution
2896 specifies a namespace, all of whose definitions are considered to be exported. By default,
2897 definitions are not exported.

2898 The SCA artifact export is useful for SCA domains containing heterogeneous mixtures of
2899 contribution packagings and technologies, where artifact-related or packaging-related
2900 mechanisms are unlikely to work across different kinds of contribution.

- 2901 • **namespace (required)** – For XML definitions, which are identified by QNames, the
2902 namespace should be the namespace URI for the exported definitions. For XML
2903 technologies that define multiple *symbol spaces* that can be used within one namespace
2904 (e.g. WSDL port types are a different symbol space from WSDL bindings), all definitions
2905 from all symbol spaces are exported.

2906 Technologies that use naming schemes other than QNames must use a different export
2907 element from the same substitution group as the the SCA <export> element. The
2908 element used identifies the technology, and may use any value for the namespace that is
2909

appropriate for that technology. For example, <export.java> can be used can be used to export java definitions, in which case the namespace should be a fully qualified package name.

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

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

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

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

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

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

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

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

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

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

1.10.2.3 Contribution Packaging using ZIP

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

2961 metadata specified by the section 'SCA Contribution Metadata Document' be present.
2962 Specifically, it may contain a top-level "META-INF" directory and a "META-INF/sca-
2963 contribution.xml" file and there may also be an optional "META-INF/sca-contribution-
2964 generated.xml" file in the package. SCA defined artifacts as well as non-SCA defined artifacts
2965 such as object files, WSDL definition, Java classes may be present anywhere in the ZIP archive,
2966 A up to date definition of the ZIP file format is published by PKWARE in [an Application Note on](#)
2967 [the .ZIP file format \[12\]](#).

2968

2969 **1.10.3 Installed Contribution**

2970 As noted in the section above, the contents of a contribution should not need to be modified in
2971 order to install and use it within a domain. An *installed contribution* is a contribution with all of
2972 the associated information necessary in order to execute *deployable composites* within the
2973 contribution.

2974 An installed contribution is made up of the following things:

- 2975 • Contribution Packaging – the contribution that will be used as the starting point for
2976 resolving all references
- 2977 • Contribution base URI
- 2978 • Dependent contributions: a set of snapshots of other contributions that are used to
2979 resolve the import statements from the root composite and from other dependent
2980 contributions
 - 2981 ○ Dependent contributions may or may not be shared with other installed
2982 contributions.
 - 2983 ○ When the snapshot of any contribution is taken is implementation defined, ranging
2984 from the time the contribution is installed to the time of execution
- 2985 • Deployment-time composites.
2986 These are composites that are added into an installed contribution after it has been
2987 deployed. This makes it possible to provide final configuration and access to
2988 implementations within a contribution without having to modify the contribution. These
2989 are optional, as composites that already exist within the contribution may also be used for
2990 deployment.

2991

2992 Installed contributions provide a context in which to resolve qualified names (e.g. QNames in
2993 XML, fully qualified class names in Java).

2994 If multiple dependent contributions have exported definitions with conflicting qualified names,
2995 the algorithm used to determine the qualified name to use is implementation dependent.
2996 Implementations of SCA may also generate an error if there are conflicting names.

2997

2998 **1.10.3.1 Installed Artifact URIs**

2999 When a contribution is installed, all artifacts within the contribution are assigned URIs, which are
3000 constructed by starting with the base URI of the contribution and adding the relative URI of each
3001 artifact (recalling that SCA requires that any packaging format be able to offer up its artifacts in
3002 a single hierarchy).

3003

3004 **1.10.4 Operations for Contributions**

3005 SCA Domains provide the following conceptual functionality associated with contributions
3006 (meaning the function may not be represented as addressable services and also meaning that

3007 equivalent functionality may be provided in other ways). The functionality is optional meaning
3008 that some SCA runtimes may choose not to provide that functionality in any way:

3009 **1.10.4.1 install Contribution & update Contribution**

3010
3011 Creates or updates an installed contribution with a supplied root contribution, and installed at a
3012 supplied base URI. A supplied dependent contribution list specifies the contributions that should
3013 be used to resolve the dependencies of the root contribution and other dependent contributions.
3014 These override any dependent contributions explicitly listed via the location attribute in the
3015 import statements of the contribution.

3016
3017 SCA follows the simplifying assumption that the use of a contribution for resolving anything also
3018 means that all other exported artifacts can be used from that contribution. Because of this, the
3019 dependent contribution list is just a list of installed contribution URIs. There is no need to specify
3020 what is being used from each one.

3021 Each dependent contribution is also an installed contribution, with its own dependent
3022 contributions. By default these dependent contributions of the dependent contributions (which
3023 we will call *indirect dependent contributions*) are included as dependent contributions of the
3024 installed contribution. However, if a contribution in the dependent contribution list exports any
3025 conflicting definitions with an indirect dependent contribution, then the indirect dependent
3026 contribution is not included (i.e. the explicit list overrides the default inclusion of indirect
3027 dependent contributions). Also, if there is ever a conflict between two indirect dependent
3028 contributions, then the conflict must be resolved by an explicit entry in the dependent
3029 contribution list.

3030 Note that in many cases, the dependent contribution list can be generated. In particular, if a
3031 domain is careful to avoid creating duplicate definitions for the same qualified name, then it is
3032 easy for this list to be generated by tooling.

3033 **1.10.4.2 add Deployment Composite & update Deployment Composite**

3034 Adds or updates a deployment composite using a supplied composite ("composite by value" – a
3035 data structure, not an existing resource in the domain) to the contribution identified by a
3036 supplied contribution URI. The added or updated deployment composite is given a relative URI
3037 that matches the @name attribute of the composite, with a ".composite" suffix. Since all
3038 composites must run within the context of a installed contribution (any component
3039 implementations or other definitions are resolved within that contribution), this functionality
3040 makes it possible for the deployer to create a composite with final configuration and wiring
3041 decisions and add it to an installed contribution without having to modify the contents of the root
3042 contribution.

3043 Also, in some use cases, a contribution may include only implementation code (e.g. PHP scripts).
3044 It should then be possible for those to be given component names by a (possibly generated)
3045 composite that is added into the installed contribution, without having to modify the packaging.

3046 **1.10.4.3 remove Contribution**

3047 Removes the deployed contribution identified by a supplied contribution URI.

3048

3049 **1.10.5 Use of Existing (non-SCA) Mechanisms for Resolving Artifacts**

3050

3051 For certain types of artifact, there are existing and commonly used mechanisms for referencing a
3052 specific concrete location where the artifact can be resolved.

3053 Examples of these mechanisms include:

- 3054 • For WSDL files, the *@wsdlLocation* attribute is a hint that has a URI value pointing to
3055 the place holding the WSDL itself.

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

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

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

Note: If one of these 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.

1.10.6 Domain-Level Composite

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

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

1.10.6.1 *add To Domain-Level Composite*

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

1.10.6.2 *remove From Domain-Level Composite*

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

1.10.6.3 *get Domain-Level Composite*

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

1.10.6.4 *get QName Definition*

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

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

2 Appendix 1

2.1 XML Schemas

2.1.1 sca.xsd

```
3112 <?xml version="1.0" encoding="UTF-8"?>
3113 <!-- (c) Copyright SCA Collaboration 2006 -->
3114 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3115         targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
3116         xmlns:sca="http://www.osoa.org/xmlns/sca/1.0">
3117
3118     <include schemaLocation="sca-core.xsd"/>
3119
3120     <include schemaLocation="sca-interface-java.xsd"/>
3121     <include schemaLocation="sca-interface-wsdl.xsd"/>
3122
3123     <include schemaLocation="sca-implementation-java.xsd"/>
3124     <include schemaLocation="sca-implementation-composite.xsd"/>
3125
3126     <include schemaLocation="sca-binding-webservice.xsd"/>
3127     <include schemaLocation="sca-binding-jms.xsd"/>
3128     <include schemaLocation="sca-binding-sca.xsd"/>
3129
3130     <include schemaLocation="sca-definitions.xsd"/>
3131     <include schemaLocation="sca-policy.xsd"/>
3132
3133 </schema>
```

2.1.2 sca-core.xsd

```
3137 <?xml version="1.0" encoding="UTF-8"?>
3138 <!-- (c) Copyright SCA Collaboration 2006, 2007 -->
3139 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3140         targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
3141         xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
3142         elementFormDefault="qualified">
3143
3144     <element name="componentType" type="sca:ComponentType"/>
3145     <complexType name="ComponentType">
3146         <sequence>
3147             <element ref="sca:implementation" minOccurs="0" maxOccurs="1"/>
3148             <choice minOccurs="0" maxOccurs="unbounded">
3149                 <element name="service" type="sca:ComponentService" />
3150                 <element name="reference" type="sca:ComponentReference"/>
3151                 <element name="property" type="sca:Property"/>
3152             </choice>
3153             <any namespace="##other" processContents="lax" minOccurs="0"
3154                 maxOccurs="unbounded"/>
3155         </sequence>
3156         <attribute name="constrainingType" type="QName" use="optional"/>
3157         <anyAttribute namespace="##any" processContents="lax"/>

```

```

3158 </complexType>
3159
3160 <element name="composite" type="sca:Composite"/>
3161 <complexType name="Composite">
3162   <sequence>
3163     <element name="include" type="anyURI" minOccurs="0"
3164       maxOccurs="unbounded"/>
3165     <choice minOccurs="0" maxOccurs="unbounded">
3166       <element name="service" type="sca:Service"/>
3167       <element name="property" type="sca:Property"/>
3168       <element name="component" type="sca:Component"/>
3169       <element name="reference" type="sca:Reference"/>
3170       <element name="wire" type="sca:Wire"/>
3171     </choice>
3172     <any namespace="##other" processContents="lax" minOccurs="0"
3173       maxOccurs="unbounded"/>
3174   </sequence>
3175   <attribute name="name" type="NCName" use="required"/>
3176   <attribute name="targetNamespace" type="anyURI" use="required"/>
3177   <attribute name="local" type="boolean" use="optional" default="false"/>
3178   <attribute name="autowire" type="boolean" use="optional" default="false"/>
3179   <attribute name="constrainingType" type="QName" use="optional"/>
3180   <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3181   <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3182   <anyAttribute namespace="##any" processContents="lax"/>
3183 </complexType>
3184
3185 <complexType name="Service">
3186   <sequence>
3187     <element ref="sca:interface" minOccurs="0" maxOccurs="1" />
3188     <element name="operation" type="sca:Operation" minOccurs="0"
3189       maxOccurs="unbounded" />
3190     <choice minOccurs="0" maxOccurs="unbounded">
3191       <element ref="sca:binding" />
3192       <any namespace="##other" processContents="lax"
3193         minOccurs="0" maxOccurs="unbounded" />
3194     </choice>
3195     <element ref="sca:callback" minOccurs="0" maxOccurs="1" />
3196     <any namespace="##other" processContents="lax" minOccurs="0"
3197       maxOccurs="unbounded" />
3198   </sequence>
3199   <attribute name="name" type="NCName" use="required" />
3200   <attribute name="promote" type="anyURI" use="required" />
3201   <attribute name="requires" type="sca:listOfQNames" use="optional" />
3202   <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3203   <anyAttribute namespace="##any" processContents="lax" />
3204 </complexType>
3205
3206 <element name="interface" type="sca:Interface" abstract="true" />
3207 <complexType name="Interface" abstract="true"/>
3208
3209 <complexType name="Reference">
3210   <sequence>
3211     <element ref="sca:interface" minOccurs="0" maxOccurs="1" />
3212     <element name="operation" type="sca:Operation" minOccurs="0"
3213       maxOccurs="unbounded" />
3214     <choice minOccurs="0" maxOccurs="unbounded">
3215       <element ref="sca:binding" />
3216       <any namespace="##other" processContents="lax" />
3217     </choice>

```

```

3218     <element ref="sca:callback" minOccurs="0" maxOccurs="1" />
3219     <any namespace="##other" processContents="lax" minOccurs="0"
3220         maxOccurs="unbounded" />
3221 </sequence>
3222 <attribute name="name" type="NCName" use="required" />
3223 <attribute name="target" type="sca:listOfAnyURIs" use="optional"/>
3224 <attribute name="wiredByImpl" type="boolean" use="optional" default="false"/>
3225 <attribute name="multiplicity" type="sca:Multiplicity"
3226     use="optional" default="1..1" />
3227 <attribute name="promote" type="sca:listOfAnyURIs" use="required" />
3228 <attribute name="requires" type="sca:listOfQNames" use="optional" />
3229 <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3230 <anyAttribute namespace="##any" processContents="lax" />
3231 </complexType>
3232
3233 <complexType name="SCAPropertyBase" mixed="true">
3234     <!-- mixed="true" to handle simple type -->
3235     <sequence>
3236         <any namespace="##any" processContents="lax" minOccurs="0"
3237             maxOccurs="1" />
3238         <!-- NOT an extension point; This xsd:any exists to accept
3239             the element-based or complex type property
3240             i.e. no element-based extension point under "sca:property" -->
3241     </sequence>
3242 </complexType>
3243
3244 <!-- complex type for sca:property declaration -->
3245 <complexType name="Property" mixed="true">
3246     <complexContent>
3247         <extension base="sca:SCAPropertyBase">
3248             <!-- extension defines the place to hold default value -->
3249             <attribute name="name" type="NCName" use="required"/>
3250             <attribute name="type" type="QName" use="optional"/>
3251             <attribute name="element" type="QName" use="optional"/>
3252             <attribute name="many" type="boolean" default="false"
3253                 use="optional"/>
3254             <attribute name="mustSupply" type="boolean" default="false"
3255                 use="optional"/>
3256             <anyAttribute namespace="##any" processContents="lax"/>
3257             <!-- an extension point ; attribute-based only -->
3258         </extension>
3259     </complexContent>
3260 </complexType>
3261
3262 <complexType name="PropertyValue" mixed="true">
3263     <complexContent>
3264         <extension base="sca:SCAPropertyBase">
3265             <attribute name="name" type="NCName" use="required"/>
3266             <attribute name="type" type="QName" use="optional"/>
3267             <attribute name="element" type="QName" use="optional"/>
3268             <attribute name="many" type="boolean" default="false"
3269                 use="optional"/>
3270             <attribute name="source" type="string" use="optional"/>
3271             <attribute name="file" type="anyURI" use="optional"/>
3272             <anyAttribute namespace="##any" processContents="lax"/>
3273             <!-- an extension point ; attribute-based only -->
3274         </extension>
3275     </complexContent>
3276 </complexType>
3277

```

```

3278 <element name="binding" type="sca:Binding" abstract="true"/>
3279 <complexType name="Binding" abstract="true">
3280   <sequence>
3281     <element name="operation" type="sca:Operation" minOccurs="0"
3282       maxOccurs="unbounded" />
3283   </sequence>
3284   <attribute name="uri" type="anyURI" use="optional"/>
3285   <attribute name="name" type="NCName" use="optional"/>
3286   <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3287   <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3288 </complexType>
3289
3290 <element name="bindingType" type="sca:BindingType"/>
3291 <complexType name="BindingType">
3292   <sequence minOccurs="0" maxOccurs="unbounded">
3293     <any namespace="##other" processContents="lax" />
3294   </sequence>
3295   <attribute name="type" type="QName" use="required"/>
3296   <attribute name="alwaysProvides" type="sca:listOfQNames" use="optional"/>
3297   <attribute name="mayProvide" type="sca:listOfQNames" use="optional"/>
3298   <anyAttribute namespace="##any" processContents="lax"/>
3299 </complexType>
3300
3301 <element name="callback" type="sca:Callback"/>
3302 <complexType name="Callback">
3303   <choice minOccurs="0" maxOccurs="unbounded">
3304     <element ref="sca:binding"/>
3305     <any namespace="##other" processContents="lax"/>
3306   </choice>
3307   <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3308   <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3309   <anyAttribute namespace="##any" processContents="lax"/>
3310 </complexType>
3311
3312 <complexType name="Component">
3313   <sequence>
3314     <element ref="sca:implementation" minOccurs="0" maxOccurs="1"/>
3315     <choice minOccurs="0" maxOccurs="unbounded">
3316       <element name="service" type="sca:ComponentService"/>
3317       <element name="reference" type="sca:ComponentReference"/>
3318       <element name="property" type="sca:PropertyValue" />
3319     </choice>
3320     <any namespace="##other" processContents="lax" minOccurs="0"
3321       maxOccurs="unbounded"/>
3322   </sequence>
3323   <attribute name="name" type="NCName" use="required"/>
3324   <attribute name="autowire" type="boolean" use="optional" default="false"/>
3325   <attribute name="constrainingType" type="QName" use="optional"/>
3326   <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3327   <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3328   <anyAttribute namespace="##any" processContents="lax"/>
3329 </complexType>
3330
3331 <complexType name="ComponentService">
3332   <complexContent>
3333     <restriction base="sca:Service">
3334       <sequence>
3335         <element ref="sca:interface" minOccurs="0" maxOccurs="1"/>
3336         <element name="operation" type="sca:Operation" minOccurs="0"
3337           maxOccurs="unbounded" />

```



```

3338         <choice minOccurs="0" maxOccurs="unbounded">
3339             <element ref="sca:binding"/>
3340             <any namespace="##other" processContents="lax"
3341                 minOccurs="0" maxOccurs="unbounded"/>
3342         </choice>
3343         <element ref="sca:callback" minOccurs="0" maxOccurs="1"/>
3344         <any namespace="##other" processContents="lax" minOccurs="0"
3345             maxOccurs="unbounded"/>
3346     </sequence>
3347     <attribute name="name" type="NCName" use="required"/>
3348     <attribute name="requires" type="sca:listOfQNames"
3349         use="optional"/>
3350     <attribute name="policySets" type="sca:listOfQNames"
3351         use="optional"/>
3352     <anyAttribute namespace="##any" processContents="lax"/>
3353 </restriction>
3354 </complexContent>
3355 </complexType>
3356
3357 <complexType name="ComponentReference">
3358     <complexContent>
3359         <restriction base="sca:Reference">
3360             <sequence>
3361                 <element ref="sca:interface" minOccurs="0" maxOccurs="1" />
3362                 <element name="operation" type="sca:Operation" minOccurs="0"
3363                     maxOccurs="unbounded" />
3364                 <choice minOccurs="0" maxOccurs="unbounded">
3365                     <element ref="sca:binding" />
3366                     <any namespace="##other" processContents="lax" />
3367                 </choice>
3368                 <element ref="sca:callback" minOccurs="0" maxOccurs="1" />
3369                 <any namespace="##other" processContents="lax" minOccurs="0"
3370                     maxOccurs="unbounded" />
3371             </sequence>
3372             <attribute name="name" type="NCName" use="required" />
3373             <attribute name="autowire" type="boolean" use="optional"
3374                 default="false"/>
3375             <attribute name="wiredByImpl" type="boolean" use="optional"
3376                 default="false"/>
3377             <attribute name="target" type="sca:listOfAnyURIs" use="optional"/>
3378             <attribute name="multiplicity" type="sca:Multiplicity"
3379                 use="optional" default="1..1" />
3380             <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3381             <attribute name="policySets" type="sca:listOfQNames"
3382                 use="optional"/>
3383             <anyAttribute namespace="##any" processContents="lax" />
3384         </restriction>
3385     </complexContent>
3386 </complexType>
3387
3388 <element name="implementation" type="sca:Implementation"
3389     abstract="true" />
3390 <complexType name="Implementation" abstract="true">
3391     <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3392     <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3393 </complexType>
3394
3395 <element name="implementationType" type="sca:ImplementationType"/>
3396 <complexType name="ImplementationType">
3397     <sequence minOccurs="0" maxOccurs="unbounded">

```

```

3398     <any namespace="##other" processContents="lax" />
3399 </sequence>
3400 <attribute name="type" type="QName" use="required"/>
3401 <attribute name="alwaysProvides" type="sca:listOfQNames" use="optional"/>
3402 <attribute name="mayProvide" type="sca:listOfQNames" use="optional"/>
3403 <anyAttribute namespace="##any" processContents="lax"/>
3404 </complexType>
3405
3406 <complexType name="Wire">
3407   <sequence>
3408     <any namespace="##other" processContents="lax" minOccurs="0"
3409       maxOccurs="unbounded"/>
3410   </sequence>
3411   <attribute name="source" type="anyURI" use="required"/>
3412   <attribute name="target" type="anyURI" use="required"/>
3413   <anyAttribute namespace="##any" processContents="lax"/>
3414 </complexType>
3415
3416 <element name="include" type="sca:Include"/>
3417 <complexType name="Include">
3418   <attribute name="name" type="QName"/>
3419   <anyAttribute namespace="##any" processContents="lax"/>
3420 </complexType>
3421
3422 <complexType name="Operation">
3423   <attribute name="name" type="NCName" use="required"/>
3424   <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3425   <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3426   <anyAttribute namespace="##any" processContents="lax"/>
3427 </complexType>
3428
3429 <element name="constrainingType" type="sca:ConstrainingType"/>
3430 <complexType name="ConstrainingType">
3431   <sequence>
3432     <choice minOccurs="0" maxOccurs="unbounded">
3433       <element name="service" type="sca:ComponentService"/>
3434       <element name="reference" type="sca:ComponentReference"/>
3435       <element name="property" type="sca:Property" />
3436     </choice>
3437     <any namespace="##other" processContents="lax" minOccurs="0"
3438       maxOccurs="unbounded"/>
3439   </sequence>
3440   <attribute name="name" type="NCName" use="required"/>
3441   <attribute name="targetNamespace" type="anyURI"/>
3442   <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3443   <anyAttribute namespace="##any" processContents="lax"/>
3444 </complexType>
3445
3446
3447 <simpleType name="Multiplicity">
3448   <restriction base="string">
3449     <enumeration value="0..1"/>
3450     <enumeration value="1..1"/>
3451     <enumeration value="0..n"/>
3452     <enumeration value="1..n"/>
3453   </restriction>
3454 </simpleType>
3455
3456 <simpleType name="OverrideOptions">
3457   <restriction base="string">

```

```

3458         <enumeration value="no"/>
3459         <enumeration value="may"/>
3460         <enumeration value="must"/>
3461     </restriction>
3462 </simpleType>
3463
3464 <!-- Global attribute definition for @requires to permit use of intents
3465     within WSDL documents -->
3466 <attribute name="requires" type="sca:listOfQNames"/>
3467
3468 <!-- Global attribute definition for @endsConversation to mark operations
3469     as ending a conversation -->
3470 <attribute name="endsConversation" type="boolean" default="false"/>
3471
3472 <simpleType name="listOfQNames">
3473     <list itemType="QName"/>
3474 </simpleType>
3475
3476 <simpleType name="listOfAnyURIs">
3477     <list itemType="anyURI"/>
3478 </simpleType>
3479
3480 </schema>

```

3481 2.1.3 sca-binding-sca.xsd

```

3482
3483 <?xml version="1.0" encoding="UTF-8"?>
3484 <!-- (c) Copyright SCA Collaboration 2006, 2007 -->
3485 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3486     targetNamespace="http://www.oesa.org/xmlns/sca/1.0"
3487     xmlns:sca="http://www.oesa.org/xmlns/sca/1.0"
3488     elementFormDefault="qualified">
3489
3490 <include schemaLocation="sca-core.xsd"/>
3491
3492 <element name="binding.sca" type="sca:SCABinding"
3493     substitutionGroup="sca:binding"/>
3494 <complexType name="SCABinding">
3495     <complexContent>
3496         <extension base="sca:Binding">
3497             <sequence>
3498                 <element name="operation" type="sca:Operation" minOccurs="0"
3499                     maxOccurs="unbounded" />
3500             </sequence>
3501             <attribute name="uri" type="anyURI" use="optional"/>
3502             <attribute name="name" type="QName" use="optional"/>
3503             <attribute name="requires" type="sca:listOfQNames"
3504                 use="optional"/>
3505             <attribute name="policySets" type="sca:listOfQNames"
3506                 use="optional"/>
3507             <anyAttribute namespace="##any" processContents="lax"/>
3508         </extension>
3509     </complexContent>
3510 </complexType>
3511 </schema>
3512

```

3513 2.1.4 sca-interface-java.xsd

3514

```

3515 <?xml version="1.0" encoding="UTF-8"?>
3516 <!-- (c) Copyright SCA Collaboration 2006 -->
3517 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3518   targetNamespace="http://www.osea.org/xmlns/sca/1.0"
3519   xmlns:sca="http://www.osea.org/xmlns/sca/1.0"
3520   elementFormDefault="qualified">
3521
3522   <include schemaLocation="sca-core.xsd"/>
3523
3524   <element name="interface.java" type="sca:JavaInterface"
3525     substitutionGroup="sca:interface"/>
3526   <complexType name="JavaInterface">
3527     <complexContent>
3528       <extension base="sca:Interface">
3529         <sequence>
3530           <any namespace="##other" processContents="lax" minOccurs="0"
3531             maxOccurs="unbounded"/>
3532         </sequence>
3533         <attribute name="interface" type="NCName" use="required"/>
3534         <attribute name="callbackInterface" type="NCName" use="optional"/>
3535         <anyAttribute namespace="##any" processContents="lax"/>
3536       </extension>
3537     </complexContent>
3538   </complexType>
3539 </schema>

```

3540

3541 2.1.5 sca-interface-wsdl.xsd

3542

```

3543 <?xml version="1.0" encoding="UTF-8"?>
3544 <!-- (c) Copyright SCA Collaboration 2006 -->
3545 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3546   targetNamespace="http://www.osea.org/xmlns/sca/1.0"
3547   xmlns:sca="http://www.osea.org/xmlns/sca/1.0"
3548   elementFormDefault="qualified">
3549
3550   <include schemaLocation="sca-core.xsd"/>
3551
3552   <element name="interface.wsdl" type="sca:WSDLPortType"
3553     substitutionGroup="sca:interface"/>
3554   <complexType name="WSDLPortType">
3555     <complexContent>
3556       <extension base="sca:Interface">
3557         <sequence>
3558           <any namespace="##other" processContents="lax" minOccurs="0"
3559             maxOccurs="unbounded"/>
3560         </sequence>
3561         <attribute name="interface" type="anyURI" use="required"/>
3562         <attribute name="callbackInterface" type="anyURI" use="optional"/>
3563         <anyAttribute namespace="##any" processContents="lax"/>
3564       </extension>
3565     </complexContent>
3566   </complexType>
3567 </schema>

```

3568

3569 2.1.6 sca-implementation-java.xsd

3570

```

3571 <?xml version="1.0" encoding="UTF-8"?>

```

```

3572 <!-- (c) Copyright SCA Collaboration 2006 -->
3573 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3574     targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
3575     xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
3576     elementFormDefault="qualified">
3577
3578     <include schemaLocation="sca-core.xsd"/>
3579
3580     <element name="implementation.java" type="sca:JavaImplementation"
3581         substitutionGroup="sca:implementation"/>
3582     <complexType name="JavaImplementation">
3583         <complexContent>
3584             <extension base="sca:Implementation">
3585                 <sequence>
3586                     <any namespace="##other" processContents="lax"
3587                         minOccurs="0" maxOccurs="unbounded"/>
3588                 </sequence>
3589                 <attribute name="class" type="NCName" use="required"/>
3590                 <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3591                 <attribute name="policySets" type="sca:listOfQNames"
3592                     use="optional"/>
3593                 <anyAttribute namespace="##any" processContents="lax"/>
3594             </extension>
3595         </complexContent>
3596     </complexType>
3597 </schema>

```

2.1.7 sca-implementation-composite.xsd

```

3598
3599
3600 <?xml version="1.0" encoding="UTF-8"?>
3601 <!-- (c) Copyright SCA Collaboration 2006 -->
3602 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3603     targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
3604     xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
3605     elementFormDefault="qualified">
3606
3607     <include schemaLocation="sca-core.xsd"/>
3608     <element name="implementation.composite" type="sca:SCAImplementation"
3609         substitutionGroup="sca:implementation"/>
3610     <complexType name="SCAImplementation">
3611         <complexContent>
3612             <extension base="sca:Implementation">
3613                 <sequence>
3614                     <any namespace="##other" processContents="lax" minOccurs="0"
3615                         maxOccurs="unbounded"/>
3616                 </sequence>
3617                 <attribute name="name" type="QName" use="required"/>
3618                 <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3619                 <attribute name="policySets" type="sca:listOfQNames"
3620                     use="optional"/>
3621                 <anyAttribute namespace="##any" processContents="lax"/>
3622             </extension>
3623         </complexContent>
3624     </complexType>
3625 </schema>
3626

```

2.1.8 sca-definitions.xsd

3628

```
3629 <?xml version="1.0" encoding="UTF-8"?>
3630 <!-- (c) Copyright SCA Collaboration 2006 -->
3631 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3632         targetNamespace="http://www.oesa.org/xmlns/sca/1.0"
3633         xmlns:sca="http://www.oesa.org/xmlns/sca/1.0"
3634         elementFormDefault="qualified">
3635
3636     <include schemaLocation="sca-core.xsd"/>
3637
3638     <element name="definitions">
3639         <complexType>
3640             <choice minOccurs="0" maxOccurs="unbounded">
3641                 <element ref="sca:intent"/>
3642                 <element ref="sca:policySet"/>
3643                 <element ref="sca:binding"/>
3644                 <element ref="sca:bindingType"/>
3645                 <element ref="sca:implementationType"/>
3646                 <any namespace="##other" processContents="lax" minOccurs="0"
3647                     maxOccurs="unbounded"/>
3648             </choice>
3649         </complexType>
3650     </element>
3651 </schema>
```

3653

3654 **2.1.9 sca-binding-webservice.xsd**

3655 Is described in [the SCA Web Services Binding specification \[9\]](#)

3656 **2.1.10sca-binding-jms.xsd**

3657 Is described in [the SCA JMS Binding specification \[11\]](#)

3658 **2.1.11sca-policy.xsd**

3659 Is described in [the SCA Policy Framework specification \[10\]](#)

2.2 SCA Concepts

2.2.1 Binding

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.

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.

2.2.2 Component

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.

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.

2.2.3 Service

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.

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.

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.

2.2.3.1 Remotable Service

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.

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

2.2.3.2 Local Service

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

3708 Local services may rely on by-reference calling conventions, or may assume a very fine-
3709 grained interaction style that is incompatible with remote distribution. They may also use
3710 technology-specific data-types.

3711 Currently a service is local only if it defined by a Java interface not marked with the
3712 @Remotable annotation.

3713

3714 **2.2.4 Reference**

3715 **SCA references** represent a dependency that an implementation has on a service that is supplied
3716 by some other implementation, where the service to be used is specified through configuration. In
3717 other words, a reference is a service that an implementation may call during the execution of its
3718 business function. References are typed by an interface.

3719 For composites, composite references can be accessed by components within the composite like any
3720 service provided by a component within the composite. Composite references can be used as the
3721 targets of wires from component references when configuring Components.

3722 A composite reference can be used to access a service such as: an SCA service provided by another
3723 SCA composite, a Web service, a stateless session EJB, a data base stored procedure or an EIS
3724 service, and so on. References use **bindings** to describe the access method used to their services.
3725 SCA provides an **extensibility mechanism** that allows the introduction of new binding types to
3726 references.

3727

3728 **2.2.5 Implementation**

3729 An implementation is concept that is used to describe a piece of software technology such as a Java
3730 class, BPEL process, XSLT transform, or C++ class that is used to implement one or more services in
3731 a service-oriented application. An SCA composite is also an implementation.

3732 Implementations define points of variability including properties that can be set and settable
3733 references to other services. The points of variability are configured by a component that uses the
3734 implementation. The specification refers to the configurable aspects of an implementation as its
3735 **componentType**.

3736 **2.2.6 Interface**

3737 **Interfaces** define one or more business functions. These business functions are provided by
3738 Services and are used by components through References. Services are defined by the Interface
3739 they implement. SCA currently supports two interface type systems:

- 3740 • Java interfaces
 - 3741 • WSDL portTypes
- 3742

3743 SCA also provides an extensibility mechanism by which an SCA runtime can add support for
3744 additional interface type systems.

3745 Interfaces may be **bi-directional**. A bi-directional service has service operations which must be
3746 provided by each end of a service communication – this could be the case where a particular service
3747 requires a “callback” interface on the client, which is calls during the process of handing service
3748 requests from the client.

3749

3750 **2.2.7 Composite**

3751 An SCA composite is the basic unit of composition within an SCA Domain. An **SCA Composite** is an
3752 assembly of Components, Services, References, and the Wires that interconnect them. Composites
3753 can be used to contribute elements to an **SCA Domain**.

3754 A **composite** has the following characteristics:

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

2.2.8 Composite inclusion

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

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

2.2.9 Property

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

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

2.2.10 Domain

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

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

2.2.11 Wire

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

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

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

3 References

3799

3800

3801 [1] SCA Java Component Implementation Specification

3802 SCA Java Common Annotations and APIs Specification

3803 http://www.osoa.org/download/attachments/35/SCA_JavaComponentImplementation_V100.pdf

3804 http://www.osoa.org/download/attachments/35/SCA_JavaAnnotationsAndAPIs_V100.pdf

3805

3806 [2] SDO Specification

3807 <http://www.osoa.org/download/attachments/36/Java-SDO-Spec-v2.1.0-FINAL.pdf>

3808

3809 [3] SCA Example Code document

3810 http://www.osoa.org/download/attachments/28/SCA_BuildingYourFirstApplication_V09.pdf

3811

3812 [4] JAX-WS Specification

3813 <http://jcp.org/en/jsr/detail?id=101>

3814

3815 [5] WS-I Basic Profile

3816 <http://www.ws-i.org/deliverables/workinggroup.aspx?wg=basicprofile>

3817

3818 [6] WS-I Basic Security Profile

3819 <http://www.ws-i.org/deliverables/workinggroup.aspx?wg=basicsecurity>

3820

3821 [7] Business Process Execution Language (BPEL)

3822 http://www.oasis-open.org/committees/documents.php?wg_abbrev=wsbpe1

3823

3824 [8] WSDL Specification

3825 WSDL 1.1: <http://www.w3.org/TR/wsd1>

3826 WSDL 2.0: <http://www.w3.org/TR/wsd120/>

3827

3828 [9] SCA Web Services Binding Specification

3829 http://www.osoa.org/download/attachments/35/SCA_WebServiceBindings_V100.pdf

3830

3831 [10] SCA Policy Framework Specification

3832 http://www.osoa.org/download/attachments/35/SCA_Policy_Framework_V100.pdf

3833

3834 [11] SCA JMS Binding Specification

3835 http://www.osoa.org/download/attachments/35/SCA_JMSBinding_V100.pdf

3836

3837 [12] ZIP Format Definition

3838 <http://www.pkware.com/documents/casestudies/APPNOTE.TXT>

3839