# SCA Service Component Architecture

## Assembly Model Specification
## Extensions for Event Processing and Pub/Sub

SCA Version 1.0 April 15 2009


| Technical Contacts: | Michael Beisiegel | IBM Corporation |
|---|---|---|
| | Vladislav Bezrukhov | SAP AG |
| | Dave Booz | IBM Corporation |
| | Martin Chapman | Oracle |
| | Mike Edwards | IBM Corporation. |
| | Anish Karmarkar | Oracle |
| | Ashok Malhotra | Oracle |
| | Peter Niblett | IBM Corporation |
| | Sanjay Patil | SAP AG |
| | Scott Vorthmann | TIBCO |

## Copyright Notice

**© Copyright Cape Clear Software, International Business Machines Corp, Interface21, IONA Technologies, Oracle, Primeton Technologies, Red Hat, SAP AG., Siemens AG., Software AG., Sun Microsystems, Inc., Sybase Inc., TIBCO Software Inc., 2005, 2009. All rights reserved.**

## License

The SCA – Assembly Model Specification Extensions for Event Processing and Pub/Sub V1.0 Specification (the "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 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 Specification, or portions thereof, that you make:

1. A link or URL to the 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 Specification.

Cape Clear Software, International Business Machines Corp., Interface21, IONA Technologies, Oracle, Primeton Technologies, Red Hat, SAP AG, Siemens AG, Software AG., Sun Microsystems Inc., Sybase Inc., TIBCO Software Inc. (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 Specification.

THE 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 Specification.

The name and trademarks of the Authors may NOT be used in any manner, including advertising or publicity pertaining to the Specification or its contents without specific, written prior permission. Title to copyright in the 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 is considered Final by the OSOA community. Feedback is no longer being solicited, but may be considered. If this specification has been provided to a standards organization for additional development or stewardship, you are encouraged to submit any comments and suggestions per that group's feedback process.

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

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.

Primeton is a registered trademark of Primeton Technologies, Ltd.

Red Hat is a registered trademark of Red Hat 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 – Event Processing and Pub/Sub Extensions ..... 1
1.1 Introduction ..... 1
1.1.1 Terminology ..... 1
1.1.2 Event Processing Overview ..... 1
1.2 Overview ..... 3
1.2.1 Diagrams used to represent SCA Artifacts ..... 3
1.2.2 Connections from Producers to Consumers ..... 5
1.2.2.1 Linking Producers to Consumers ..... 5
1.2.2.2 Producers, Consumers and Composites ..... 6
1.2.3 Event Processing Examples ..... 6
1.2.3.1 Multiple Producers linked to multiple Consumers via a Channel - within a Composite ..... 6
1.2.3.2 Producers linked to Consumers via Domain Channels ..... 8
1.2.3.3 Composite with Promotion of Producers and Consumers ..... 9
1.3 Component ..... 12
1.3.1 Example Component ..... 14
1.3.2 Declaration of Event Types on Producers and Consumers ..... 15
1.4 Implementation ..... 16
1.4.1 Component Type ..... 17
1.4.1.1 Example ComponentType ..... 19
1.4.1.2 Example Implementation ..... 19
1.5 Interface ..... 19
1.6 Composite ..... 19
1.6.1 Property – Definition and Configuration ..... 21
1.6.2 References ..... 21
1.6.3 Service ..... 21
1.6.4 Wire ..... 21
1.6.5 Using Composites as Component Implementations ..... 21
1.6.6 Using Composites through Inclusion ..... 21
1.6.7 Composites which Include Component Implementations of Multiple Types ..... 21
1.6.8 ConstrainingType ..... 21
1.6.9 Producer ..... 21

1.6.10 Consumer ........ 23
1.7 Binding ........ 24
1.8 SCA Definitions ........ 24
1.9 Extension Model ........ 24
1.10 Packaging and Deployment ........ 24
1.11 Channels ........ 24
1.11.1 Scopes of Channels ........ 25
1.11.2 The Default Domain Channel ........ 26
1.11.3 The URI of a Channel ........ 26
1.12 Representation of Events and Event Types in SCA ........ 26
1.12.1 Event Type and Associated Metadata ........ 26
1.12.2 Format of Event Type Definitions ........ 27
1.12.3 Events with No Event Type ........ 27
1.13 Filters: Selecting Subsets of Events ........ 28
1.13.1 Form of Explicit Filter Elements ........ 28
1.13.2 Event Type Filters ........ 29
1.13.2.1 Event Type Filter Examples ........ 30
1.13.3 Business Data Filters ........ 30
1.13.3.1 XPATH 1.0 Dialect ........ 30
1.13.4 Event Metadata Filters ........ 31
2 Appendix 1 ........ 32
2.1 XML Schemas ........ 32
2.1.1 sca.xsd ........ 32
2.1.2 sca-core.xsd ........ 32
2.1.3 sca-binding-sca.xsd ........ 39
2.1.4 sca-interface-java.xsd ........ 40
2.1.5 sca-interface-wsdl.xsd ........ 40
2.1.6 sca-implementation-java.xsd ........ 40
2.1.7 sca-implementation-composite.xsd ........ 40
2.1.8 sca-definitions.xsd ........ 40
2.1.9 sca-binding-webservice.xsd ........ 40
2.1.10 sca-binding-jms.xsd ........ 40
2.1.11 sca-policy.xsd ........ 40
2.1.12 sca-eventDefinition.xsd ........ 40
2.2 SCA Concepts ........ 42
3 Java Implementation Type ........ 43
3.1 Event Consumer methods ........ 43

3.2 Event Producers ..... 44
3.3 Event Types ..... 45
4 References ..... 46

# 1 Assembly Model – Event Processing and Pub/Sub Extensions

## 1.1 Introduction

This document describes the ***Event Processing and Pub/Sub Extensions for the SCA Assembly Model***, which deals with

- ***Event Processing***, which is computing that performs operations on ***events***, including creating, reading, transforming, and deleting events or event objects/representations. Event Processing components interact by creating event messages which are then distributed to other Event Processing components. An Event Processing component can, in addition, interact with other SCA components using SCA's regular service invocation mechanisms.
- ***Publication*** and ***Subscription*** (often shortened to ***Pub/Sub***), which is a particular style of organizing the components which produce and consume events in which the producing components are decoupled from the consuming components. Components that are interested in consuming events specify their interest through a subscription rather than an interface. The same event may be received by multiple subscribers.

The document starts with a description of the Event Processing and Pub/Sub extensions and then goes on to show the changes to the relevant parts of the SCA Assembly specification that are required by these extensions. This document is based on the OSOA SCA Assembly Specification V1.00 [1]. It does not incorporate any changes or additions made in the preparation of the OASIS version of the SCA Assembly specification [2].

### 1.1.1 Terminology

- *event* – a message sent to zero or more parties that contains information about a situation that has occurred
- *producer* - entity that creates events
- *consumer* - entity that receives events
- *subscription* - records a consumer's interest in receiving specific kinds of events from some location
- *source* – the place from which a consumer receives events
- *target* – the place to which a producer sends events
- *publication* – the sending of an event from a producer to some targets
- *event type* – every event instance can have an associated event type. Each event type is identified by a unique QName and has an associated shape and optionally constraints on the event instance
- *channel* –a mechanism to connect a set of producers with a set of consumers
- *filter* - a mechanism for refining the set of events received by a consumer. A filter may operate on business data within the event itself, or on metadata about the event.

### 1.1.2 Event Processing Overview

SCA defines ***service invocation*** as one mechanism through which two components communicate. With service invocation, one component, the client, invokes an operation on a service reference, which causes that operation to be invoked on a second component, the service provider. The significant charateristics of service invocation are that:

- Each invocation by the client on a reference operation causes one invocation of the operation on one service provider

- The operation itself typically has some implied semantics – the client is expecting some specific task to be performed by the service provider, possibly involving specific data being returned by the provider
- A particular operation is typically grouped with a set of other related operations, as defined by an interface, which as a whole make up the service offered by the provider. The need to implement the interface as a whole is a requirement for the code implementing the components. There is also a requirement that the complete set of operations declared on a reference is supplied by the service provider.
- The provider may respond to the operation invocation with zero or more response messages. These messages may be returned synchronously or asynchronously, but they are returned to the client component that made the original invocation. That they are returned is part of the service contract between the client and the provider

In contrast, in **event processing** applications one component, the producer, creates a message called an event, which is sent out and can be received by any number of other components, called consumers. The significant characteristics of this mechanism are that:

- Each event created by a producer may be received by zero, one or many consumer components. The producer is unaware of the specific consumers or the number of consumers that receive any event.
- The consumer cannot respond to an event received – there is in principle no knowledge of the producer component and no route provided by which a response message could be sent to it. The component receiving an event can in turn send out events, but there is no implication that the original producer component will receive any of those events.
- What is done when a consumer receives an event has no implied semantics – the consumer can do what it likes with the event and there are no semantics agreed with the producer
- There is no requirement that a consumer consumes all of the event types that can be produced by a given producer. Neither is there a requirement that a producer produces all of the event types that can be consumed by a consumer. Unlike services, there is no matching of an interface on the producer to an interface on the consumer.

  There is also no direct relationship between event types and the implementation operations or methods used to produce or consume them - eg a single operation can handle one event type or many event types, as desired by the writer of the implementation code.
- A consumer can filter which events it is prepared to accept – there is no guarantee that it actually does anything with a given event. The filtering may be on the event type or on the business data within the event or on other metadata associated with the event.

Service operations which are ***one-way*** are close in nature to the sending and receiving of events, but it is notable that for one-way service operations the client component must be aware of the number of target services (multiplicity 0..n or 1..n specified) and the client has to call the operation once for each target. For an event, the producer component simply sends a single event once through its producer – the event is sent to all the consumer components that have expressed interest in that event and are connected (including none), without the producer component being aware of the number or of the recipients.

Event processing involves more loosely-coupled method of combining components into an application than using service interfaces. Events place fewer requirements on the components at each end of the communication. Effectively, in event processing it is only the event types that are shared between the producers and the consumers.

Loose coupling is futher emphasized through the use of ***Pub/Sub***. With Pub/Sub, producers are not connected directly to any consumers – instead, a group of zero or more producers is connected with a group of zero or more consumers through a logical intermediary, called a

97 ***Channel***. The producers publish events to the channel and the consumers receive events from
98 the channel. The actual origin of an event received by a consumer can be any of the producers –
99 without the consumer being directly connected to any of the producers.

100 In SCA event processing, component implementations may have zero or more ***producers*** and
101 zero or more ***consumers***. The producers and consumers can indicate which event type(s) they
102 deal with. SCA components configure implementations to express where producer events are
103 published to and where consumer events are subscribed from.

104

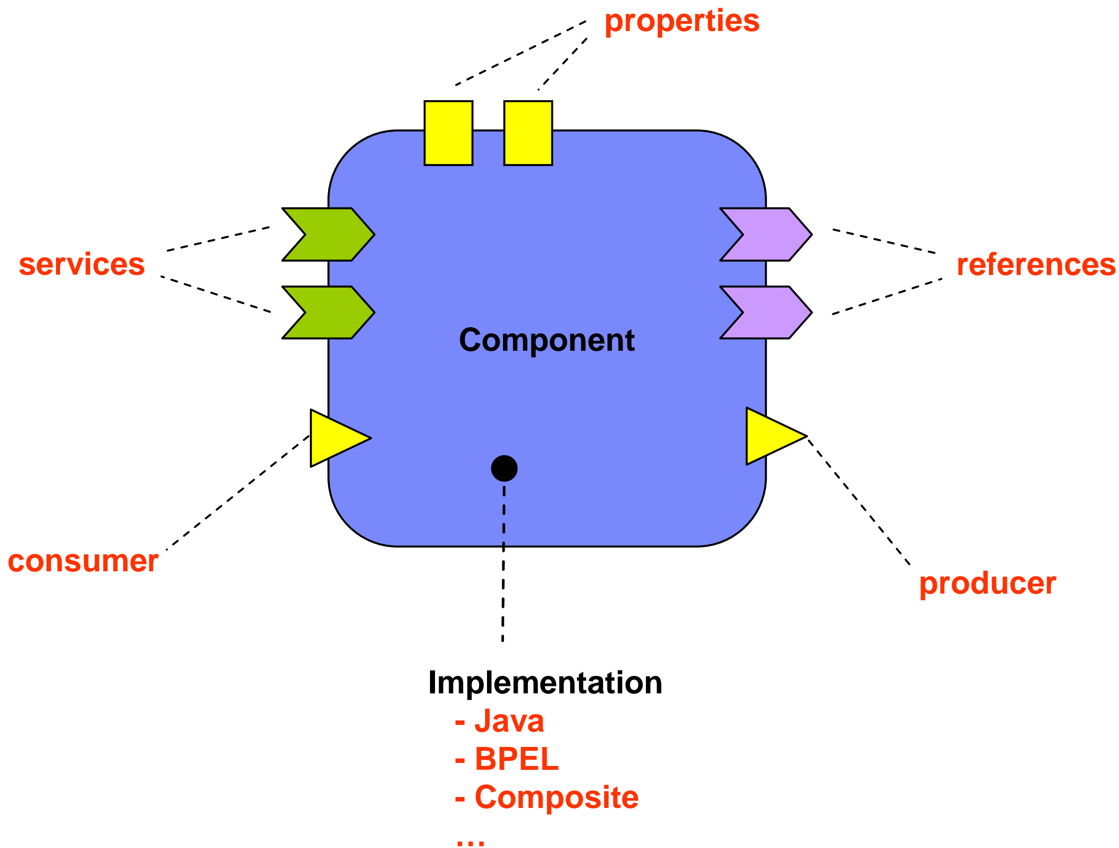
## 105 *1.2 Overview*

106 This section remains unchanged from the original OSOA SCA Assembly Specification V1.00 [1].
107

### 108 1.2.1 Diagrams used to represent SCA Artifacts

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

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

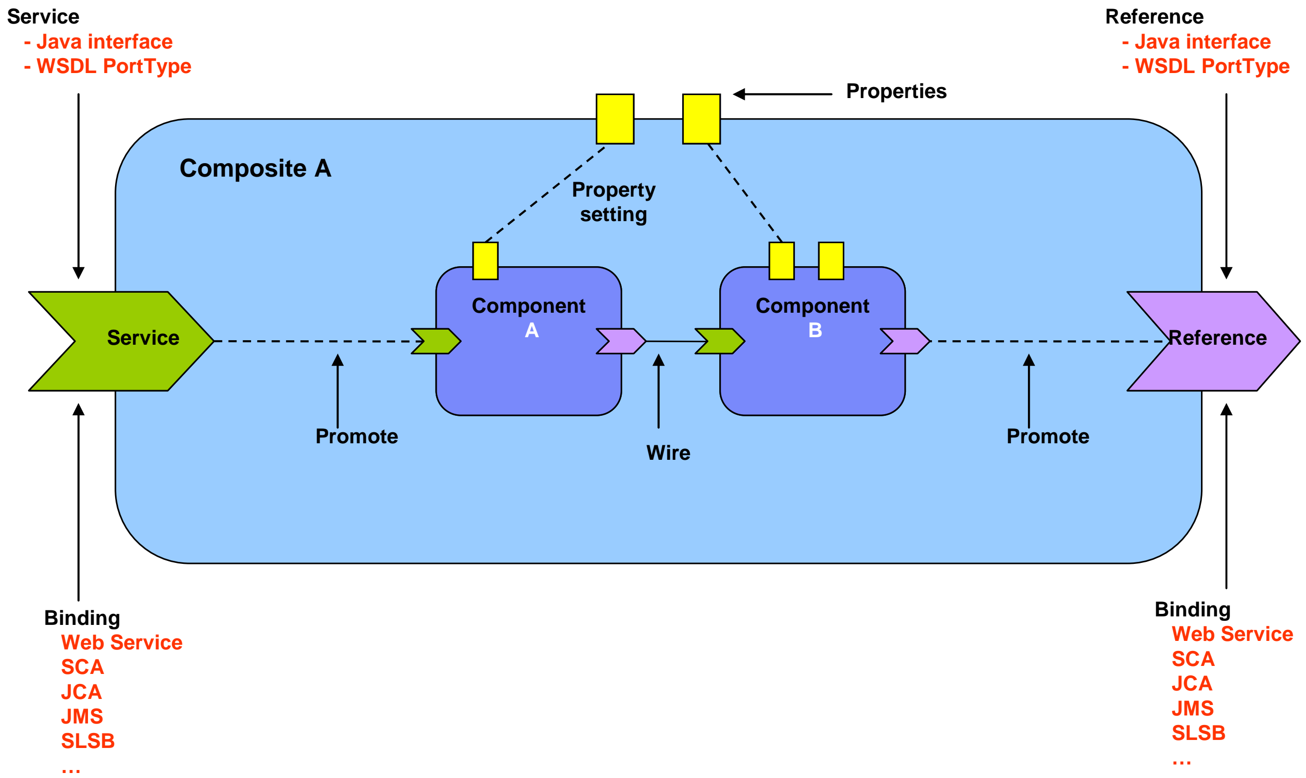



113

114 **Figure 1 SCA Component Diagram**

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

117

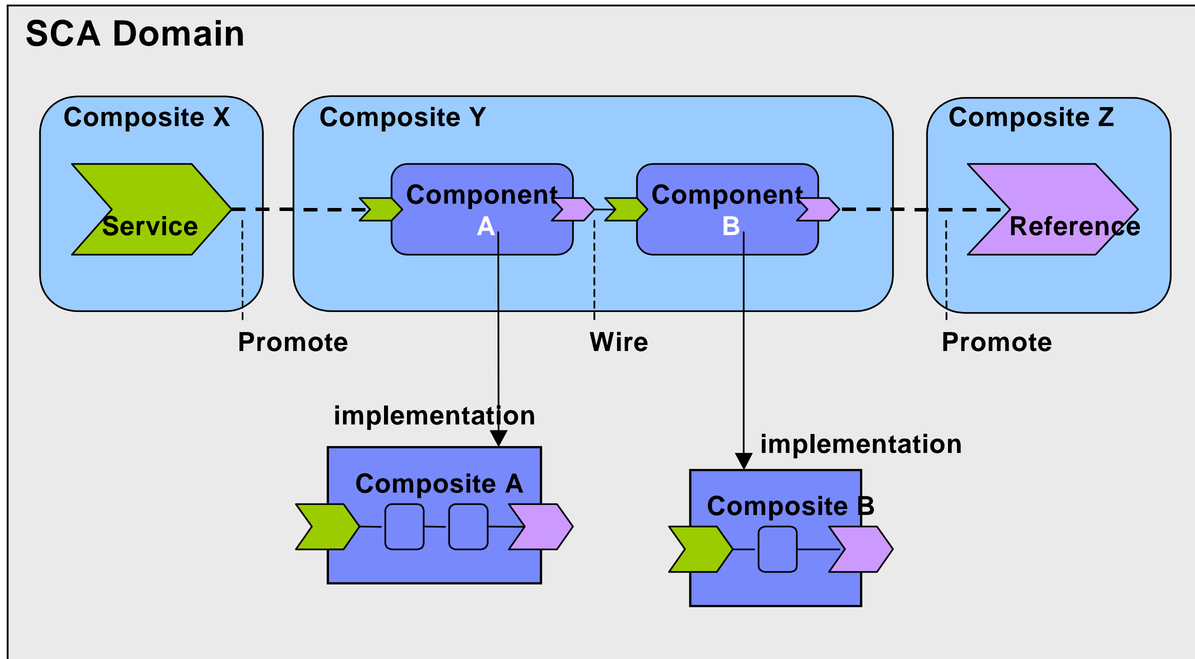



118

119 **Figure 2: SCA Composite Diagram**

120

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



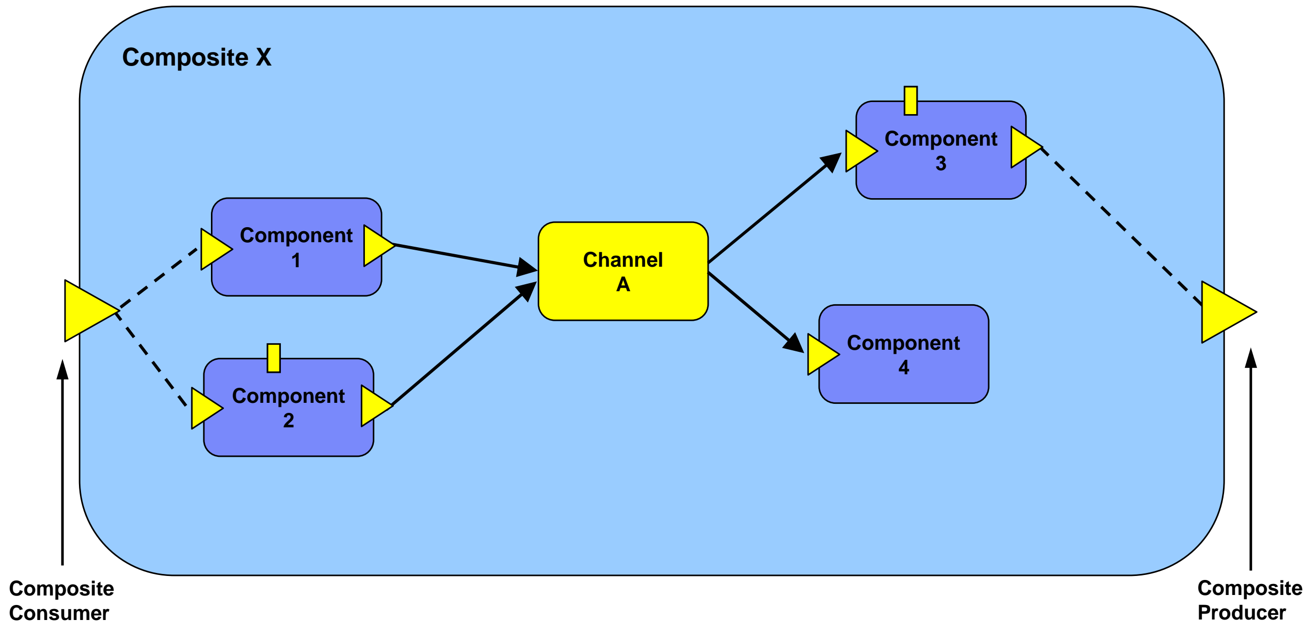

123

124 **Figure 3 SCA Domain Diagram**

125

126 The following diagram shows an SCA composite involving components that communicate using
127 event processing:

128




129

130 **Figure 14: SCA Composite Diagram with Event Processing**

131

132 ## 1.2.2 Connections from Producers to Consumers

133

134 In SCA, events flow from producers to consumers along routes that are defined by the
135 configuration of composites and the components and channels they contain. In particular,
136 components configure producers by declaring targets for the events created by the producer.
137 Components configure consumers by specifying sources for the events received by the consumer
138 and specifying the kind of events that are of interest.

139

140 ### *1.2.2.1 Linking Producers to Consumers*

141 Event producers can be linked to event consumers via a third party called a channel, where
142 producers are configured with the channel as a target and consumers are configured with the
143 channel as a source. Using this mechanism, producers and consumers are not directly
144 connected. It is also possible for the producer(s) to connect to a Domain channel (See the
145 section on Scopes of Channels) at a different time than when the consumer(s) connect to the
146 same channel.

147 A producer declares where the messages it produces are sent through a list of one or more
148 target URIs in its ***@target*** attribute. The form of the target URIs include:

149 - The URI of a channel in the same composite as the producer, in the form
150 ***channelName***

151 - The URI of of a channel at the Domain level in the form ***//channelName***

152

153 A consumer declares the sources for the messages it receives through a list of one or more
154 source URIs in its ***@source*** attribute. The form of the source URIs include:

155 - The URI of a channel in the same composite in the form ***channelName***
156 - The URI of a channel at the Domain level in the form ***//channelName***

157

158 #### *1.2.2.2 Producers, Consumers and Composites*

159

160 When an assembler creates a composite that is intended for use as an implementation, the
161 assembler can decide whether consumers and producers within the composite are visible outside
162 the scope of the composite or not.

163 The assembler can also decide on what level of control is given to the higher level component
164 that is using the composite as its implementation – i.e., the assembler can decide what appears
165 in the component type of the composite, which can then be configured by the higher level
166 component.

167 One technique which enables component producers to send events outside the composite and for
168 component consumers to receive events from outside the composite is to configure producers
169 and/or consumers of components inside the composite to use domain channels – that is,
170 channels at the Domain level. See the section on Scopes of Channels for more details on domain
171 channels. This approach "hard wires" the producers and consumers within the composite - the
172 higher level component cannot reconfigure the sources and targets.

173 An alternative technique for configuring a component producer element is to declare a
174 ***composite producer*** element which ***promotes*** the component producer. Similarly a
175 component consumer can be configured by declaring a ***composite consumer*** element which
176 ***promotes*** the component consumer. When producers and consumers are promoted in this way,
177 and the composite is used as the implementation of some higher level component, the assembler
178 of the higher level composite can control where the events flow to and from, through
179 configuration of the higher level component. This technique promotes reuse of the lower level
180 composite in different contexts.

181 Each producer and consumer can be connected to zero or more channels. If a producer is not
182 connected then any events it produces are discarded and are not received by any consumer. If a
183 consumer is not connected, then it never receives any events.

184 A Composite can contain one or more Channels. Events can be sent to a channel by producers
185 within the composite and events may be received from a channel by consumers within the
186 composite.

187

188 ### 1.2.3 Event Processing Examples

189

190 #### *1.2.3.1 Multiple Producers linked to multiple Consumers via a Channel - within a*
191 #### *Composite*

192

193 This example is of multiple component producers, which send events to multiple component
194 consumers via a Channel, which decouples the producers from the consumers. The assembly is
195 represented by the following diagram:

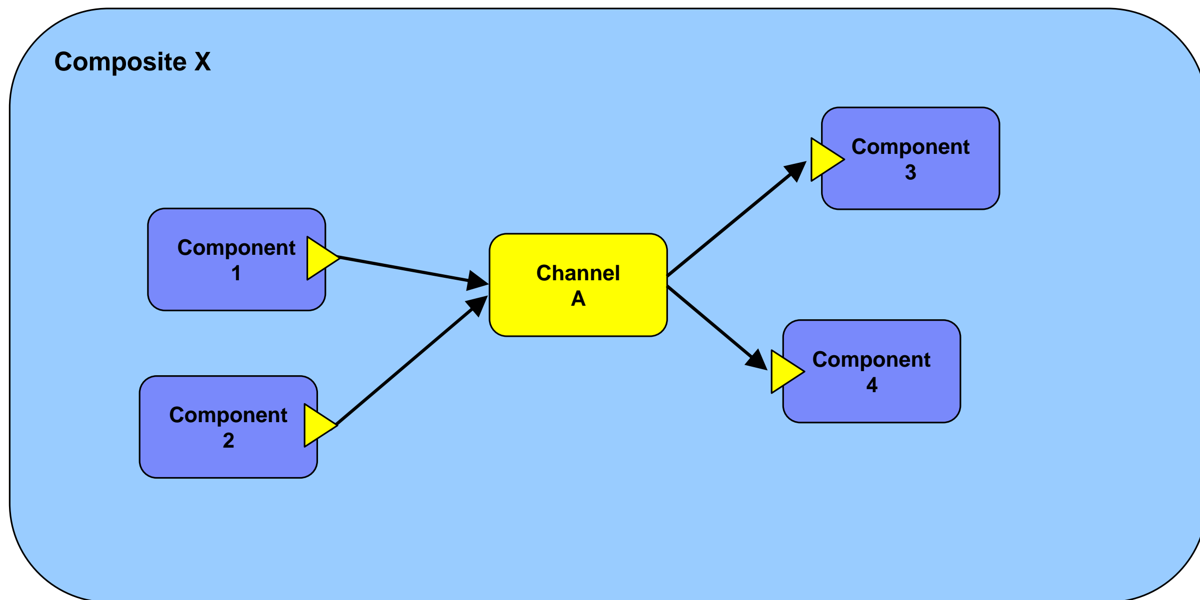



**Figure 15: Producers linked to Consumers via a local Channel**

The corresponding XML for this example follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<composite name="CompositeX"
   xmlns="http://www.osoa.org/xmlns/sca/1.0"
   targetNamespace="http://example.org/example1">

   <component name="Component1">
         <implementation.java class="org.example.Component1Impl"/>
         <producer name="Foo_Events" target="ChannelA"/>
   </component>

   <component name="Component2">
         <implementation.java class="org.example.Component2Impl"/>
         <producer name="Foo_Events" target="ChannelA"/>
   </component>

   <component name="Component3">
         <implementation.java class="org.example.Component3Impl"/>
         <consumer name="Foo_Handling" source="ChannelA"/>
   </component>

   <component name="Component4">
         <implementation.java class="org.example.Component4Impl"/>
         <consumer name="Foo_Handling" source="ChannelA"/>
   </component>

   <channel name="ChannelA"/>

</composite>
```

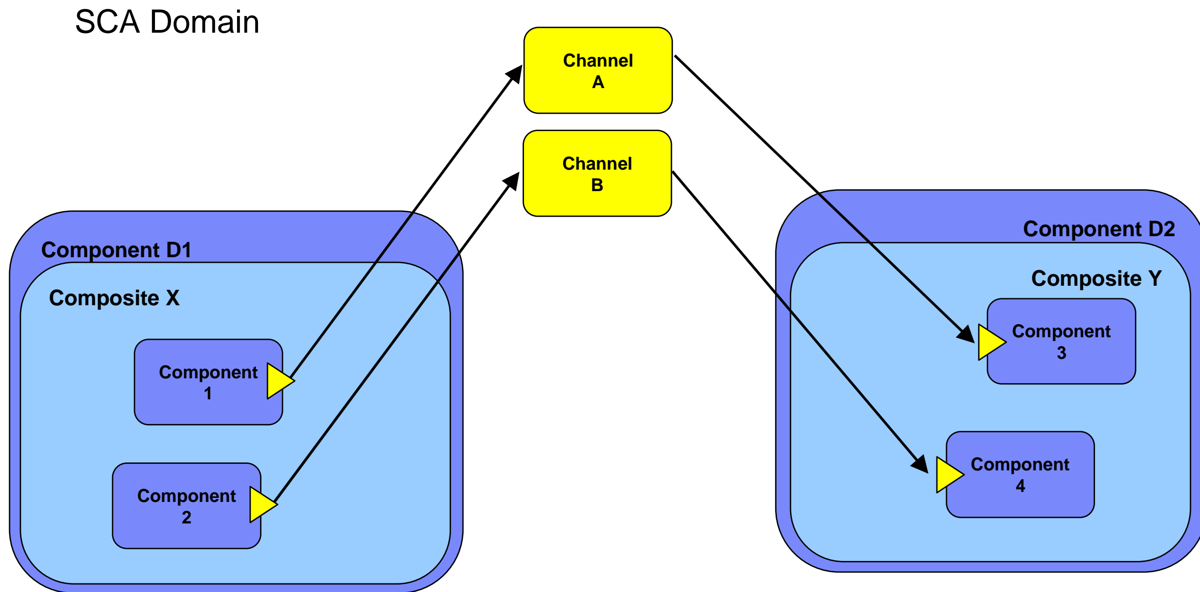
In this example, the @target attribute of the producers links them to ChannelA and the @source attribute of the consumers links them to ChannelA. All events from Component1 and Component2 are routed through the ChannelA and are sent to Component3 and Component4.

232 *1.2.3.2 Producers linked to Consumers via Domain Channels*

233

234 In this example, component producers of components nested within a domain component
235 transmit events via Domain Channels to component consumers which are also nested below the
236 domain level within a second domain component. This is represented in the following diagram:

237




238

239 **Figure 16: Producers linked to Consumers via Domain Channels**

240 The corresponding XML for this example follows:

241 For CompositeX:

```
242 <composite name="CompositeX"
243    xmlns="http://www.osoa.org/xmlns/sca/1.0"
244    targetNamespace="http://example.org/example1">
245
246    <component name="Component1">
247          <implementation.java class="org.example.Component1Impl"/>
248          <producer name="Foo_Events" target="//ChannelA"/>
249    </component>
250
251    <component name="Component2">
252          <implementation.java class="org.example.Component2Impl"/>
253          <producer name="Foo_Events" target="//ChannelB"/>
254    </component>
255
256 </composite>
```

257 For CompositeY:

```
258 <composite name="CompositeY"
259    xmlns="http://www.osoa.org/xmlns/sca/1.0"
260    targetNamespace="http://example.org/example1">
261
262    <component name="Component3">
263          <implementation.java class="org.example.Component3Impl"/>
264          <consumer name="Foo_Handling" source="//ChannelA"/>
265    </component>
266
```

```
267     <component name="Component4">
268           <implementation.java class="org.example.Component4Impl"/>
269           <consumer name="Foo_Handling" source="//ChannelB"/>
270     </component>
271
272  </composite>
```

273

274 Note the @target and @source attributes of the producers and consumers use the "//" notation
275 to indicate the connection to a channel at the domain level.

276 The following is an example of one way in which the Channels could be deployed to the Domain:

```
277  <composite name="ChannelContribution"
278     xmlns="http://www.osoa.org/xmlns/sca/1.0"
279     targetNamespace="http://example.org/example1">
280
281     <channel name="ChannelA"/>
282
283     <channel name="ChannelB"/>
284
285  </composite>
```

286

287 The following is an example of two deployment composites that could be used to deploy the two
288 domain-level components (ComponentD1 and ComponentD2):

```
289  <composite name="ComponentD1Contribution"
290     xmlns="http://www.osoa.org/xmlns/sca/1.0"
291     targetNamespace="http://example.org/example1"
292     xmlns:xmp="http://example.org/example1">
293
294     <component name="ComponentD1">
295           <implementation.composite name="xmp:CompositeX"/>
296     </component>
297  </composite>
298
299  <composite name="ComponentD2Contribution"
300     xmlns="http://www.osoa.org/xmlns/sca/1.0"
301     targetNamespace="http://example.org/example1"
302     xmlns:xmp="http://example.org/example1">
303
304     <component name="ComponentD2">
305           <implementation.composite name="xmp:CompositeY"/>
306     </component>
307
308  </composite>
```

309

310 Note that the domain level components ComponentD1 and ComponentD2 are unable to configure
311 the channels that are used as sources and targets by the components in the lower level
312 composites.

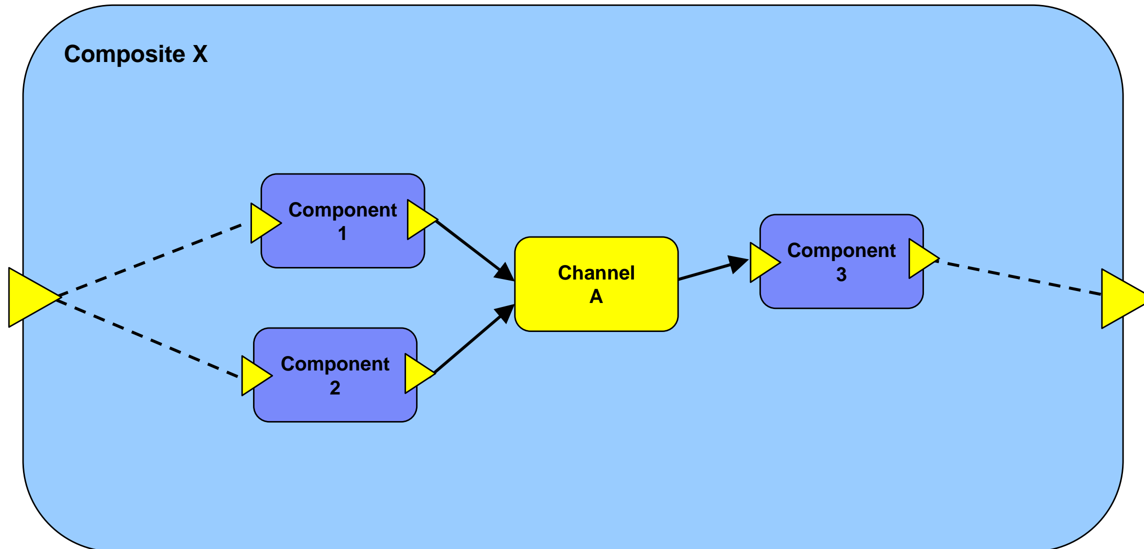
313

#### 314 *1.2.3.3 Composite with Promotion of Producers and Consumers*

315

316 This example shows how a composite can be constructed so that the composite promotes some
317 component cosumers and promotes some component producers. This is represented in the
318 following diagram:

319




320

321 **Figure 17: Promotion of Consumers and Producers by a Composite**

322 The corresponding XML for this example follows:

```
323 <?xml version="1.0" encoding="UTF-8"?>
324 <composite name="CompositeX"
325    xmlns="http://www.osoa.org/xmlns/sca/1.0"
326    targetNamespace="http://example.org/example1">
327
328    <consumer name="Bar_Handling"
329             promotes="Component1/BarHandling Component2/Bar_Handling"/>
330
331    <component name="Component1">
332          <implementation.java class="org.example.Component1Impl"/>
333          <consumer name="Bar_Handling"/>
334          <producer name="Foo_Events" target="ChannelA"/>
335    </component>
336
337    <component name="Component2">
338          <implementation.java class="org.example.Component2Impl"/>
339          <consumer name="Bar_Handling"/>
340          <producer name="Foo_Events" target="ChannelA"/>
341    </component>
342
343    <channel name="ChannelA"/>
344
345    <component name="Component3">
346          <implementation.java class="org.example.Component3Impl"/>
347          <consumer name="Foo_Handling" source="ChannelA"/>
348          <producer name="Special_Events"/>
349    </component>
350
351    <producer name="Special_Events" promotes="Component3/Special_Events"/>
352
353 </composite>
```

354

355 Here, CompositeX has a consumer element named Bar_Handling and producer element named
356 Special_Events. The Bar_Handling consumer promotes the consumers of Component1 and

357 Component2. The Special_Events producer promotes the producer of Component3.
358

359 When CompositeX is used as an implementation by a higher-level component, the consumer and
360 producer elements of the composite permit the assembler of the higher level component to
361 control where the events relating to this composite are sent to and received from, through
362 configuration of the higher level component. The Component Type of CompositeX above is as
363 follows:

```
364  <componentType  xmlns="http://www.osoa.org/xmlns/sca/1.0"
365
366     <consumer name="Bar_Handling" />
367
368     <producer name="Special_Events />
369
370  </componentType>
```

## 1.3 Component

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

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

Components are declared as subelements of a composite in an ***xxx.composite*** file. A component is represented by a ***component element*** which is a child of the composite element. There can be ***zero or more*** component elements within a composite. The following snippet shows the composite schema with the schema for the component child element. Note that this snippet, and the explanation that follows, focuses only on the additions made by this specification to the OSOA SCA Assembly Specification V1.00 [1] component schema. All existing elements/attributes in V1.00 are unaffected.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Component schema snippet -->
<composite … >

   ...

   <component … >*
         <implementation/>
         <service … />*
         <reference … />*
         <property … />*
         <consumer name="xs:NCName" source="list of xs:anyURI"?
                     requires="list of xs:QName"?
                     policySets="list of xs:QName"?>*
               <filters/>?
               <binding/>*
         </consumer>
         <producer name="xs:NCName" target="list of xs:anyURI"?
                     typeNames="list of xs:QName"?
                     typeNamespaces="list of xs:anyURI"?
                     requires="list of xs:QName"?
                     policySets="list of xs:QName"?>*
               <binding/>*
         </producer>
   </component>

   ...

</composite>
```

The component element can have ***zero or more consumer elements*** as children, which are used to configure the consumers of the component. The consumers that can be configured are defined by the implementation.

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

422 - ***name (1..1)*** – the name of the consumer. MUST be unique amongst the producer,
423 consumer, service and reference elements of the component. MUST match the name of a
424 consumer defined by the implementation.
425 - ***requires (0..1)*** – a list of policy intents. See the Policy Framework specification [10] for
426 a description of this attribute.
427 - ***policySets (0..1)*** – a list of policy sets. See the Policy Framework specification [10] for a
428 description of this attribute.
429 - ***source (0..1)*** – a list of one or more of event sources such as the URI of a channel. The
430 form of the URI for a channel is described in section The URI of a Channel.

431

432 The consumer element has the following child elements:

433 - ***binding (0..1)*** – zero or more binding elements, each of which defines a transport
434 binding which is used for the transmission of events to this consumer. If not specified, an
435 SCA default binding (binding.sca) is used.
436
437 - ***filters*** – ***(0..1)*** filter elements.
438
439 See the section Filters: Selecting Subsets of Events for a detailed description of Filters.

440 The consumer can receive events from all of the event sources identified in the @source
441 attribute. It is valid to specify no sources (ie the consumer is "unconnected"). If the consumer is
442 unconnected, no events are received.

443 The component element can have ***zero or more producer elements*** as children which are used
444 to configure the producers of the component. The producers that can be configured are defined
445 by the implementation.

446 The producer element has the following attributes:
447

448 - ***name (1..1)*** – the name of the producer. MUST be unique amongst the producer,
449 consumer, service and reference elements of the component. MUST match the name of a
450 producer defined by the implementation.
451 - ***requires (0..1)*** – a list of policy intents. See the Policy Framework specification [10] for
452 a description of this attribute.
453 - ***policySets (0..1)*** – a list of policy sets. See the Policy Framework specification [10] for a
454 description of this attribute.
455 - ***target (0..1)*** – a list of one or more of targets to which events are sent, such as the URI
456 of a channel. Where multiple targets are identified, all the messages emitted by the
457 producer are sent to each target. The form of the URI for a channel is described in section
458 The URI of a Channel.
459 - ***typeNames (0..1)*** – a list of one or more Event Type QNames which are sent by this
460 producer. Each QName in the list MUST be associated with a Namespace URI. This
461 association is performed using the namespace declarations that are in-scope where the
462 QName expression appears. Unprefixed QNames are permitted, provided there is a
463 default namespace declaration in-scope where the QName expression appears. QNames
464 that belong to no namespace are not allowed.
465 If the @typeNames attribute is omitted, the value defaults to the value of the
466 @typeNames attribute for the producer of the same name in the componentType of the
467 implementation used by the component. If the @typeNames attribute is omitted and if the
468 corresponding producer in the componentType of the implementation also does not have

469 this attribute, then the producer is unconstrainted with respect to the Even Type QNames
470 for the events that are sent by the producer. If the componentType has a value for
471 @typeNames then the value of @typeNames for the component producer element MUST
472 match that in the componentType.

473 - ***typeNamespaces (0..1)*** – a list of one or more Event Type Namespace URI for events
474 sent by this producer.

475 If the @typeNamespaces attribute is omitted, the value defaults to the value of the
476 @typeNamespaces attribute for the producer of the same name in the componentType of
477 the implementation used by the component. If the @typeNamespaces attribute is
478 omitted, and if the corresponding producer in the componentType of the implementation
479 also does not have this attribute, then the producer is unconstrainted with respect to the
480 Even Type Namespace URI for the events that are sent by the producer. If the
481 componentType has a value for @typeNamespaces then the value of @typeNamespaces
482 for the component producer element MUST match that in the componentType.

483

484 Note that both attributes @typeNames and @typeNamespaces can be used together. If both
485 attributes are specified, then the producer declares that it might send events whose Event Type
486 is either listed in the @typeNames attribute or whose Event Type belongs to one of the
487 Namespaces listed in the @typeNamespaces attribute.

488

489 The producer element has the following child elements:

490 - ***binding*** – (0..1) zero or more binding elements, each of which defines a transport
491 binding which is used for the transmission of events from this producer. If not specified,
492 an SCA default binding (binding.sca) is used.

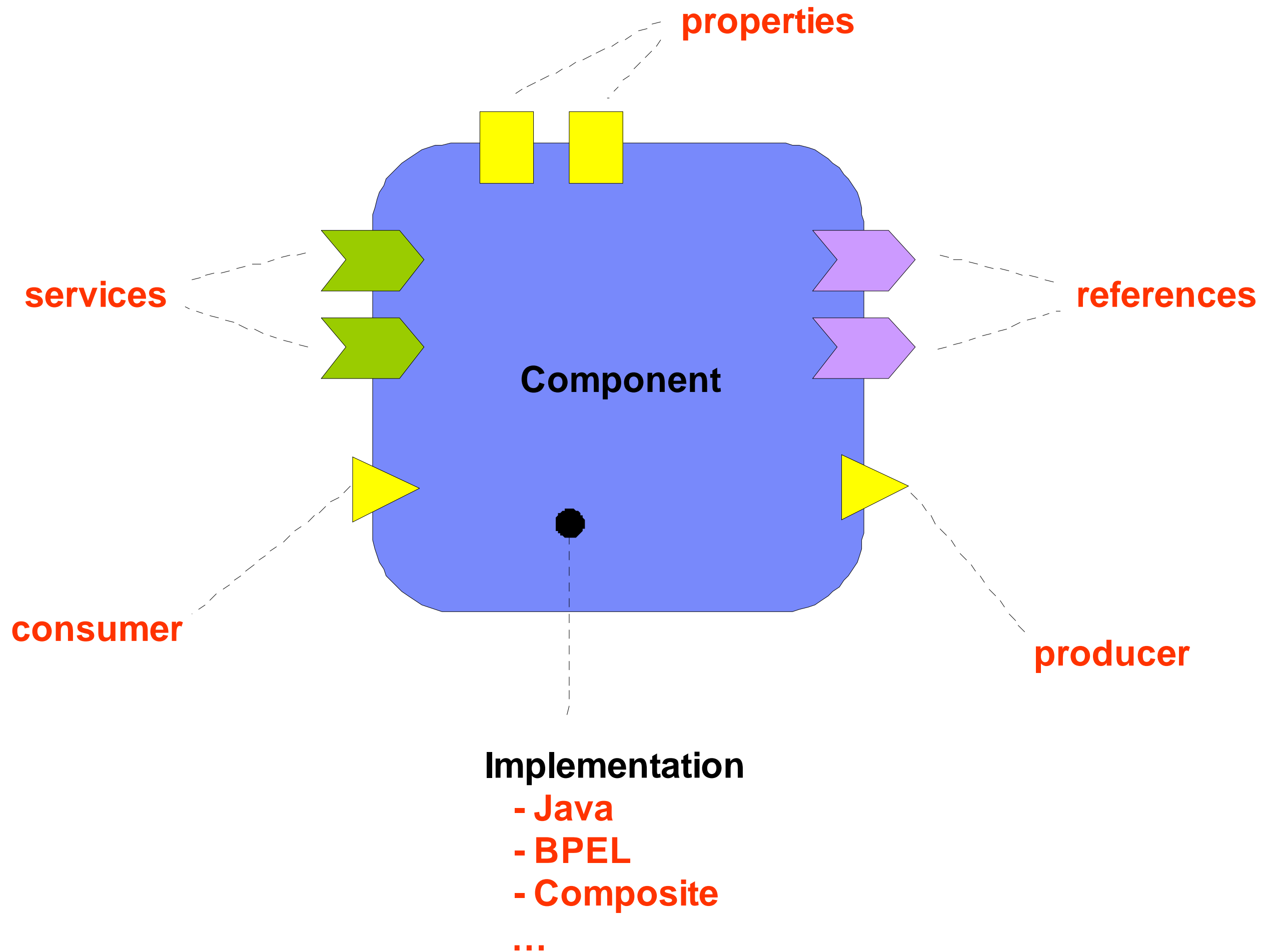
493 Events produced by the producer are sent to all the targets identified in the @target attribute. It
494 is valid to specify no targets (ie the producer is "unconnected") - in this case events produced
495 are discarded.

496

### 497 1.3.1 Example Component

498

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




501

502 **Figure 4 Component Symbol**

503 The rest of this section remains unchanged from the original OSOA SCA Assembly Specification
504 V1.00 [1].
505

### 506 1.3.2 Declaration of Event Types on Producers and Consumers

507

508 Producers can declare the set of event types that they produce through the attributes
509 producer/@typeNames and producer/@typeNamespaces. Consumers can declare the set of event
510 types that they handle by specifying a type filter. It is also possible to declare that a producer or
511 a consumer handles any event type.

512 The value of declaring the events that are produced and consumed by components and channels
513 is that:

514

515 • When the event types produced and consumed are explicitly declared, it may be possible
516 to avoid the need for runtime event filters on the consumers, providing an optimized path
517 for the handling of the events.
518

519 • Because the channel, producer and consumer declarations can include a list of event
520 types, it is possible to report an error when a producer or a consumer is connected to a

channel, where there is no chance that the produced events will be accepted by the channel or the consumer will ever get any event.

The following apply:

- A producer SHOULD only produce event types it has declared

- An SCA Runtime MAY reject events of a type from a producer which does not declare that it produces events of that type

## 1.4 Implementation

Component ***implementations*** are concrete implementations of business function. An implementation can provide ***services*** and it can make ***references*** to services provided elsewhere. An implementation can have event ***producers*** and ***consumers***. Each producer sends events of one or more event types, while each consumer receives events of one or more event types. Producers and consumers declare the set of event types that they handle through a list of event types. It is also possible to declare that a producer or a consumer handles any event type. In addition, an implementation may have some settable property values.

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

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

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

```
<implementation.java class="services.myvalue.MyValueServiceImpl"/>
<implementation.bpel process="MoneyTransferProcess"/>
<implementation.composite name="MyValueComposite"/>
```

***Services, references, consumers, producers and properties*** are the configurable aspects of an implementation. SCA refers to them collectively as the ***component type***. The characteristics of services, references and properties are described in the Component section. Depending on the implementation type, the implementation can declare the services, references, consumers, producers and properties that it has and it also can set values for the characteristics of those services, references, consumers, producers and properties.

So, for example:

- for a service, the implementation can define the interface, binding(s), a URI, intents, and policy sets, including details of the bindings

- for a reference, the implementation can define the interface, binding(s), target URI(s), intents, policy sets, including details of the bindings

567 • for a consumer, the implementation can define event filters, intents, policy sets, bindings

568 • for a property the implementation can define its type and a default value

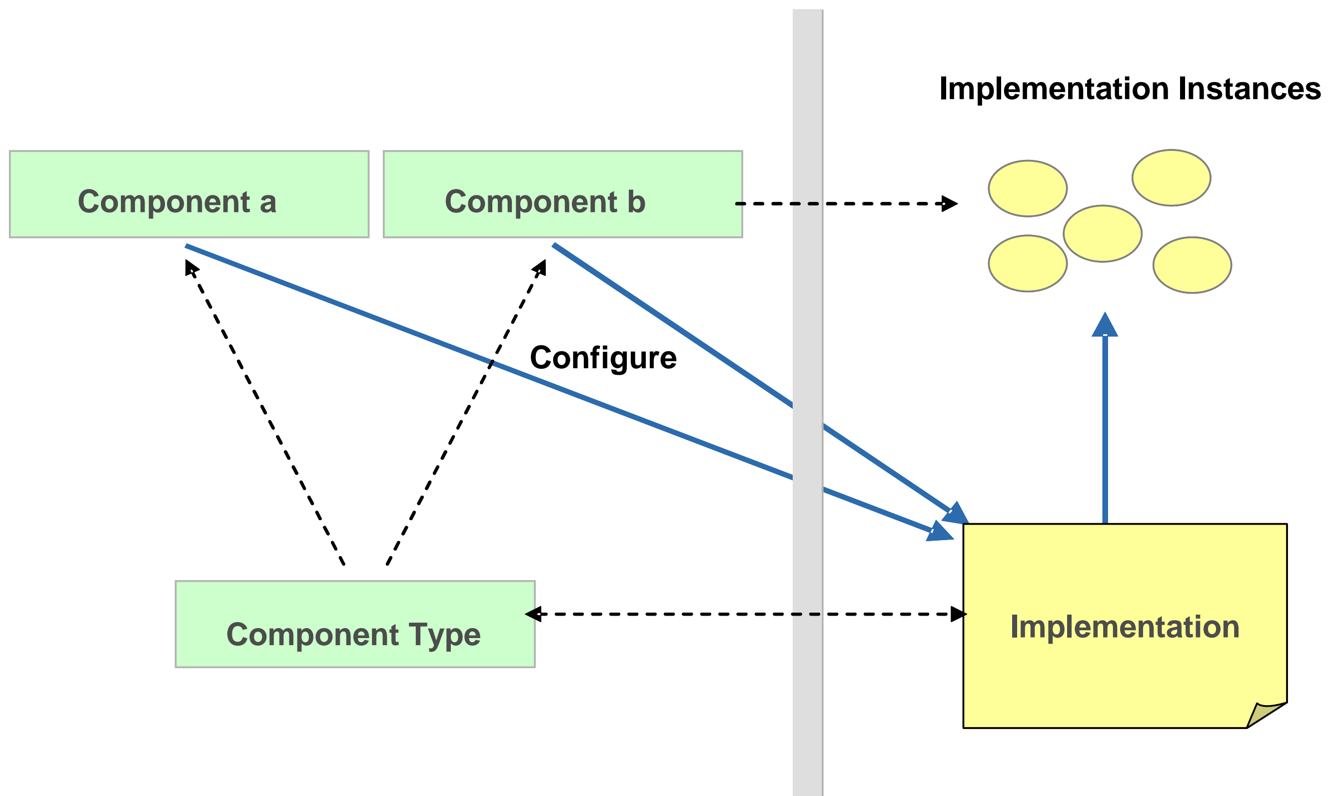
569 • the implementation itself can define intents and policy sets

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

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

578 At runtime, an ***implementation instance*** is a specific runtime instantiation of the
579 implementation – its runtime form depends on the implementation technology used. The
580 implementation instance derives its business logic from the implementation on which it is based,
581 but the values for its properties and references are derived from the component which configures
582 the implementation.

583




584

585 **Figure 6 Relationship of Component and Implementation**

586

## 587 1.4.1 Component Type

588 ***Component type*** represents the configurable aspects of an implementation. A component type
589 consists of services that are offered, references to other services that can be wired, consumers
590 to which events are delivered, producers that send out events and properties that can be set.
591 The settable properties, the settable references to services and the settable consumers and
592 prodcers are configured by a component which uses the implementation.

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

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

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

The componentType element contains zero or more Service elements, zero or more Reference elements, zero or more Consumer elements, zero or more Producer element and zero or more Property elements.

The following snippet shows the componentType schema.

```
<?xml version="1.0" encoding="ASCII"?>
<!-- Component type schema snippet -->
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0"
   constrainingType="QName"? >

   <service .../>*
   <reference .../>*
   <property .../>*
   <consumer name="xs:NCName"
         requires="list of xs:QName"?
         policySets="list of xs:QName"?>*
         <filters/>?
         <binding uri="xs:anyURI"? requires="list of xs:QName"?
               policySets="list of xs:QName"?/>*
   </consumer>
   <producer name="xs:NCName"
         typeNames="list of xs:QName"?
         typeNamespaces="list of xs:anyURI"?
         requires="list of xs:QName"?
         policySets="list of xs:QName"?>*
         <binding/>*
   </producer>

   <implementation .../>?

</componentType>
```

The ComponentType element has a single attribute:

- ***constrainingType (optional)*** – the name of a constrainingType. When specified, the set of services, references and properties of the implementation, plus related intents, is

645 constrained to the set defined by the constrainingType. See the ConstrainingType Section
646 for more details.

647 The ComponentType element has the following child elements:

648 • ***Service (0..n) -*** represents an addressable service interface of the implementation. See
649 the Service section for details.

650 • ***Reference (0..n) -*** represents a requirement that the implementation has on a service
651 provided by another component. See the Reference section for details.

652 • ***Consumer (0..n) -*** represents a place where events can be delivered to the
653 implementation. See the Component section for details.

654 • ***Producer (0..n) -*** represents a place where the implementation produces events that
655 can be sent to other components. See the Component section for details.

656 • ***Properties (0..n) -*** allow for the configuration of an implementation with externally set
657 values. See the Property section for details.

658 • ***Implementation (0..1) -*** represents characteristics inherent to the implementation
659 itself, in particular intents and policies. See the Policy Framework specification [10] for a
660 description of intents and policies.

661

662 #### *1.4.1.1 Example ComponentType*

663

664 This section remains unchanged from the original OSOA SCA Assembly Specification V1.00 [1].
665

666 #### *1.4.1.2 Example Implementation*

667

668 This section remains unchanged from the original OSOA SCA Assembly Specification V1.00 [1].
669

670 ## *1.5 Interface*

671

672 This section remains unchanged from the original OSOA SCA Assembly Specification V1.00 [1].
673

674 ## *1.6 Composite*

675

676 In this section, the only change introduced is the addition of<consumer>, <producer>, and
677 <channel> elements to the schema for the composite element. Everything else in this section
678 remains unchanged from the original OSOA SCA Assembly Specification V1.00 [1].
679

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

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

686

```
687 <?xml version="1.0" encoding="ASCII"?>
688 <!-- Composite schema snippet -->
```

```
689  <composite ... >
690
691     <include ... />*
692     <service ... />*
693     <reference ... />*
694     <property ... />*
695     <component ... />*
696     <wire ... />*
697     <consumer ... />*
698     <producer ... />*
699     <channel ... />*
700
701  </composite>
```

702

703 Composites contain ***zero or more properties, services, consumers, producers,***
704 ***components, channels, references, wires and included composites***. These artifacts are
705 described in detail in the following sections.

706 Components contain configured implementations which hold the business logic of the composite.
707 The components offer services and require references to other services and they send out events
708 via producers and receive events through consumers.

709 Channels within the composite represent intermediaries transmitting events from producers to
710 consumers entirely within the composite.

711 Composite services define the public services provided by the composite, which can be accessed
712 from outside the composite. Composite references represent dependencies which the composite
713 has on services provided elsewhere, outside the composite. Composite consumers define public
714 locations where events are received from outside the composite. Composite producers represent
715 places where the composite as a whole sends out events.

716 Wires describe the connections between component services and component references within
717 the composite. Included composites contribute the elements they contain to the using composite.

718 Composite services involve the ***promotion*** of one service of one of the components within the
719 composite, which means that the composite service is actually provided by one of the
720 components within the composite. Composite references involve the ***promotion*** of one or more
721 references of one or more components. Multiple component references can be promoted to the
722 same composite reference, as long as all the component references are compatible with one
723 another. Where multiple component references are promoted to the same composite reference,
724 then they all share the same configuration, including the same target service(s).

725 Composite consumers involve the ***promotion*** of one or more contained component consumers.
726 Composite producers involve the ***promotion*** of one or more contained component producers.

727 Composite services and composite references can use the configuration of their promoted
728 services and references respectively (such as Bindings and Policy Sets). Alternatively composite
729 services and composite references can override some or all of the configuration of the promoted
730 services and references, through the configuration of bindings and other aspects of the
731 composite service or reference.

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

735 Component producers can be promoted to composite producers and can be configured to send
736 events to other targets at the same time. Similarly, component consumers can be promoted to
737 composite consumers and can be configured to receive events from other sources at the same
738 time.

739

740 ### 1.6.1 Property – Definition and Configuration

741

742 This section remains unchanged from the original OSOA SCA Assembly Specification V1.00 [1].
743

744 ### 1.6.2 References

745

746 This section remains unchanged from the original OSOA SCA Assembly Specification V1.00 [1].
747

748 ### 1.6.3 Service

749

750 This section remains unchanged from the original OSOA SCA Assembly Specification V1.00 [1].
751

752 ### 1.6.4 Wire

753

754 This section remains unchanged from the original OSOA SCA Assembly Specification V1.00 [1].
755

756 ### 1.6.5 Using Composites as Component Implementations

757

758 This section remains unchanged from the original OSOA SCA Assembly Specification V1.00 [1].
759

760 ### 1.6.6 Using Composites through Inclusion

761

762 This section remains unchanged from the original OSOA SCA Assembly Specification V1.00 [1].
763

764 ### 1.6.7 Composites which Include Component Implementations of Multiple Types

765

766 This section remains unchanged from the original OSOA SCA Assembly Specification V1.00 [1].
767

768 ### 1.6.8 ConstrainingType

769

770 This section remains unchanged from the original OSOA SCA Assembly Specification V1.00 [1].
771

772 ### 1.6.9 Producer

773

774 The ***producers of a composite*** are defined by promoting producers defined by components
775 contained in the composite. Producers are promoted by means of a composite ***producer***
776 ***element***, which is a child element of the composite element. Promotion of the component
777 producer allows the configuration of the composite producer set by a higher level component to
778 override the configuration of the lower component producer.

779 Every event sent by any of the promoted producers is sent out by the composite producer.

780 The following snippet shows the schema for a composite producer element:

781
```
782     <producer name="xs:NCName"          promotes="list of xs:anyURI"
783         requires="list of xs:QName"?
784         policySets="list of xs:QName"?
785         typeNames="list of xs:QName"?
786         typeNamespaces="list of xs:anyURI"?>*
787         <binding uri="xs:anyURI"? requires="list of xs:QName"?
788             policySets="list of xs:QName"?/>*
789     </producer>
790
```

791 The producer element has the following ***attributes***:

- 792 ***name (required)*** – the name of the producer. MUST be unique amongst the producer
793 elements of the composite. The name MAY be different from the name of any of the
794 promoted producers.
- 795 ***requires (optional)*** – a list of policy intents. See the Policy Framework specification [10]
796 for a description of this attribute.
- 797 ***policySets (optional)*** – a list of policy sets. See the Policy Framework specification [10]
798 for a description of this attribute.
- 799 ***promotes (required)*** – identifies the promoted producers. The value is a list containing
800 entries of the form ***componentName/producerName***. The producer name is optional if
801 the component only has one producer.
- 802 ***typeNames (0..1)*** – a list of one or more Event Type QNames which are sent by this
803 producer. Each QName in the list MUST be associated with a Namespace URI. This
804 association is performed using the namespace declarations that are in-scope where the
805 QName expression appears. Unprefixed QNames are permitted, provided there is a
806 default namespace declaration in-scope where the QName expression appears. QNames
807 that belong to no namespace are not allowed.

  808 If the @typeNames attribute is omitted, the value defaults to the value of the
809 @typeNames attribute, if present, of the associated component producer or the
810 componentType of the component producer that is promoted.

  811 If the associated component producer or the componentType producer has a value for
812 @typeNames then the value of @typeNames, if present, for the composite producer
813 element MUST match that in the component propducer or the componentType producer.
- 814 ***typeNamespaces (0..1)*** – a list of one or more Event Type Namespace URI for events
815 sent by this producer.

  816 If the @typeNamespaces attribute is omitted, the value defaults to the value of the
817 @typeNamespaces attribute, if present, of the associated component producer or the
818 componentType of the component producer that is promoted.

  819 If the associated component producer or the componentType producer has a value for
820 @typeNamespaces then the value of @typeNamespaces, if present, for the composite
821 producer element MUST match that in the component producer or the componentType
822 producer.

823

824 Note that both attributes @typeNames and @typeNamespaces can be used together. If both
825 attributes are specified, then the producer declares that it might send events whose Event Type
826 is either listed in the @typeNames attribute or whose Event Type belongs to one of the
827 Namespaces listed in the @typeNamespaces attribute.

828

829 The producer element has the following ***child element***:

830 - ***binding (0..n)*** – zero or more binding elements. Each element defines a transport
831 binding which is used for the transmission of events from this producer. If not specified,
832 an SCA default binding (binding.sca) is used.

833 If bindings are specified they ***override*** the bindings defined for the promoted component
834 producers from the composite producer perspective. The bindings defined on the component
835 producers are still in effect for local transmission of messages within the composite. Details of
836 the binding element are described in the Bindings section.

837

838 ### 1.6.10 Consumer

839

840 The ***consumers of a composite*** are defined by promoting consumers defined by components
841 contained in the composite. Consumers are promoted by means of a composite ***consumer***
842 ***element***, which is a child element of the composite element. Promotion of the component
843 consumer allows the configuration of the composite consumer set by a higher level component to
844 override the configuration of the lower component consumer.

845 Every event received by the composite producer is sent on to all of the promoted consumers.

846 The following snippet shows the schema for a composite consumer element:

```
847
848    <consumer name="xs:NCName"
849          promotes="list of xs:anyURI"
850          requires="list of xs:QName"?
851          policySets="list of xs:QName"?>*
852          <filters/>?
853          <binding uri="xs:anyURI"? requires="list of xs:QName"?
854                policySets="list of xs:QName"?/>*
855    </consumer>
856
```

857 The consumer element sas the following ***attributes***:

858 - ***name (required)*** – the name of the consumer. MUST be unique amongst the consumer
859 elements of the composite. The name MAY be different from the name of any of the
860 promoted consumers.

861 - ***requires (optional)*** – a list of policy intents. See the Policy Framework specification [10]
862 for a description of this attribute.

863 - ***policySets (optional)*** – a list of policy sets. See the Policy Framework specification [10]
864 for a description of this attribute.

865 - ***promote (required)*** – identifies the promoted consumers. The value is a list containing
866 entries of the form ***componentName/consumerName***. The consumer name is optional
867 if the component only has one consumer.

868

869 The consumer element has the following ***child elements***:

870 - ***binding (0..n)*** – zero or more binding elements. Each element defines a transport
871 binding which is used for the transmission of events to this consumer. If not specified, an
872 SCA default binding (binding.sca) is used.
873

874 - ***filter (0..1)*** –filter elements
875 See the section Filters: Selecting Subsets of Events for a detailed description of Filters.

876 If bindings are specified they ***override*** any bindings defined for the promoted consumers from
877 the composite producer perspective. The bindings defined on the component consumers are still
878 in effect for local transmission of messages within the composite. Details of the binding element
879 are described in the Bindings section.

880

## 881 1.7 Binding

882

883 This section remains unchanged from the original OSOA SCA Assembly Specification V1.00 [1]
884 except bindings are also used by producers, consumers and channels, in addition to services and
885 references. Producers, consumers and channels use bindings to describe the mechanism used to
886 send and receive events.

887 A binding is defined by a ***binding element*** which is a child element of a service, a reference, a
888 producer, a consumer in a composite or in a component, or a channel element in a composite.

889

## 890 1.8 SCA Definitions

891

892 This section remains unchanged from the original OSOA SCA Assembly Specification V1.00 [1].

## 893 1.9 Extension Model

894

895 This section remains unchanged from the original OSOA SCA Assembly Specification V1.00 [1].

## 896 1.10 Packaging and Deployment

897

898 This section remains unchanged from the original OSOA SCA Assembly Specification V1.00 [1].
899

## 900 1.11 Channels

901

902 A ***channel*** is an SCA artifact that is used to connect a set of event producers to a set of event
903 consumers. The channel can accept events sent by many producers and it can send all of these
904 events to each of the set of consumers, which are subscribed to the channel.

905 One role of the channel is to act as an intermediary between the set of producers and the set of
906 consumers. The channel exists separately from any individual producer or consumer.

907 A channel acts as if it has a single consumer element with the name "in", to which producers can
908 send events. A channel acts as if it has a single producer element with the name "out", from
909 which subscribers receive events.

910 A channel may be configured with filters, which defines the set of events that the channel
911 accepts. If an event does not match the filters defined, the event is discarded

912

913 The pseudo-schema for Channels is shown here:

914

```
915
916     <channel name="xs:NCName"
917           requires="list of xs:QName"?
```

```
918            policySets="list of xs:QName"?>*
919            <filters/>?
920            <binding/>*
921      </channel>
922
923
```

924 The channel element has the following ***attributes***:
925

926 - ***name (required)*** – a name for the channel element, which must be unique amongst the
927 channel and component elements of a given composite.
928
929 - ***requires*** – (optional) a list of one or more policy ***intents*** which apply to the handling of
930 messages by this channel. See the Policy Framework specification [10] for a description of
931 this attribute.
932
933 - ***policySets*** – (optional) a list of the names of one or more policy sets which apply to the
934 handling of messages by this channel. See the Policy Framework specification [10] for a
935 description of this attribute.
936

937 The channel element has the following ***child elements***:

938

939 - ***filters*** – (0..n) zero or more filter elements, each of which defines a filter expression in a
940 particular dialect.
941
942 See the section Filters: Selecting Subsets of Events for a detailed description of Filters.
943
944 - ***binding (0..n)*** – zero or more binding elements. Each element defines a transport binding
945 which is used for the transmission of events to this channel. If not specified, an SCA default
946 binding (binding.sca) is used.
947

948 ### 1.11.1 Scopes of Channels

949

950 Channels can exist either at the Domain level or they can exist within a composite used as an
951 implementation.

952 Channels at the Domain level (i.e., channels that are present in the domain-level composite) are
953 termed ***domain channels***. They can be used as targets for producers at any level within the
954 composition hierarchy. They can be used as sources for consumers at any level within the
955 composition hierarchy. SCA runtimes MUST support the use of domain channels. To create a
956 Domain Channel, deploy a composite containing a channel directly to the SCA Domain (i.e., do
957 not use that composite as the implementation of some component in the Domain).

958 Channels within a composite used as an implementation are private to the components within
959 that composite. These ***private channels*** can only be the targets for producers existing within
960 the same composite as the channel. Private channels can only be sources for consumers existing
961 withing the same composite as the channel. SCA runtimes MAY support the use of private
962 channels - in other words, this capability is optional.

963 This division of Channels into global channels and private channels permits the assembler of an
964 application to control the set of components involved in event exchange, if required. Producers
965 and consumers of global channels are effectively uncontrolled – they exist at the Domain and
966 they can be added or removed at any time through deployment actions. Private channels have

restricted sets of producers and consumers – these sets are decided by the assembler when the composite containing them is created.

### 1.11.2 The Default Domain Channel

In SCA Event processing, there is a special ***default channel*** which is a domain channel.

The default channel always exists, even if it not declared explicitly in the configuration of the Domain. The default channel has the URI "//".

Producers and consumers at any level in the Domain can communicate using the default channel by using the URI "//" in their target or source attribute respectively.

### 1.11.3 The URI of a Channel

When used for the source of a consumer or for the target of a producer, a channel is referenced by a URI. The URI of a channel is built from the name of the channel.

The URI of a private channel is the name of the channel.

The URI of a domain channel is "//" followed by the name of the channel.

The URI of the default domain channel is simply "//".

## 1.12 Representation of Events and Event Types in SCA

Events in SCA MAY have an ***event type*** associated with them. Each event type is identified by a unique ***event type name***.

An event can have no event type metadata associated with it - for example, this can be the case for events which are created by pre-existing non-SCA event sources.

SCA has a canonical ***representation*** of event types in terms of XML and of event shapes in terms of XML schema. SCA event shapes are ***describable*** using XML infoset, although they don't have to be described using XML Schema – other type systems can be used. SCA events can have a wire format that is not XML.

Events can also have programming language specific representations. The details of the mapping between language specific formats and XML are defined by the SCA implementation language specifications.

### 1.12.1 Event Type and Associated Metadata

In SCA, ***event types*** consist of the following:

1. a unique ***event type QName***
2. a set of business data. This data is also called the ***shape*** of the event. It is possible that the same shape is used by multiple event types.
3. optional additional metadata associated with events of this type, such as creation time, and is separate from the event business data.

1010 The shape of the event should be defined in terms of an existing type system. Examples include
1011 XSD and Java.

1012 For event shape defined using XSD, this is done in terms of an XML global element, which is in
1013 turn of some XSD simple or complex type.

1014

### 1015 1.12.2 Format of Event Type Definitions

1016

1017 SCA event types are defined in event definitions files and take the following form:

1018

```
1019 <?xml version="1.0" encoding="ASCII"?>
1020 <eventDefinitions    xmlns="http://www.osoa.org/xmlns/sca/1.0"
1021          targetNamespace="xs:anyURI">
1022
1023    <sca:eventType name="xs:NCName"? ... >*
1024        ...
1025    </sca:eventType>
1026
1027 </eventDefinitions>
```

1028

1029 and, in particular, for an eventType defined using XSD:

1030

```
1031 <sca:eventType.xsd name="xs:NCName"? element="xs:QName" ...>
1032
1033    ...
1034 <sca:eventType.xsd>
```

1035

1036 where ***attributes***:

1037 - ***name (0..1)*** - the event type name, which is qualified by the target namespace of the
1038 definitions element. If absent, the event type name uses the name of the global element
1039 referenced by the @element attribute (but not its namespace)
1040 - ***element (1..1)*** - the global XSD element which defines the shape of the event

1041

1042 ***any (0..n)*** - extensibility elements allowing for additional metadata relating to the event type

1043

1044 an example of an event type definition follows:

1045

```
1046 <eventDefinitions xmlns="http://www.osoa.org/xmlns/sca/1.0"
1047              xmlns:foo="http://foo.com"
1048              targetNamespace="http://foo.com">
1049     <sca:eventType.xsd name="PrinterEvent" element="foo:PrinterEvent"/>
1050 </eventDefinitions>
1051
```

### 1052 1.12.3 Events with No Event Type

1053

1054 Events MAY have no event type metadata associated with them.

1055 From an SCA perspective (and in particular, when dealing with events of this kind in Filter
1056 statements), such events are given the special event type name of sca:NULL (a QName). This

1057 special event type name MUST NOT be used in event instances for its type metadata. It is
1058 reserved for use in composite files (such as in type filters on comsumer and type declarations on
1059 producers) to identify event instances that do not have any type metadata.

1060

## 1061 1.13 Filters: Selecting Subsets of Events

1062

1063 Event filters are used to select subsets of events from an event source. Event filters can be
1064 specified on consumers and on channels, and are then applied to the event instances that would
1065 otherwise be received by those consumers or channels.

1066 Filters can operate against various sorts of data relating to an event instance:

1067 • Event types

1068 • Event business data

1069 • Other event metadata

1070 The mechanism for expressing filters is extensible, so that in the future filters can be added that
1071 operate against other data, such as Properties of the Event channel.

1072 Filters can be expressed in a variety of dialects of filter language. It is possible to use different
1073 filter language dialects for different types of data - eg Event Metadata vs Business Data. It is
1074 possible to specify multiple filters (of the same type or different types) on a single consumer or
1075 channel.

1076 Each filter expression must resolve to a boolean where "false" means that the event instance is
1077 discarded and "true" means that the event instance is passed by the filter. Where multiple filters
1078 are present, they are logically "AND"ed together so that only messages that pass all of the filters
1079 are passed by the collection of filters.

1080 Filters can be specified on a Component consumer, Composite consumer, or a consumer in the
1081 Component Type of an implementation. All filter expressions specified on a consumer, regardless
1082 of where (Component Type, Component or Composite) they are specified are logically “AND”ed
1083 together.

1084 Filters have no side effects and filters have no state. They are evaluated against a particular
1085 event instance and indicate whether the event passes the filter or not – there are no other
1086 implications. This means that the order in which multiple filters are applied does not matter – the
1087 same result occurs whatever the order.

1088

### 1089 1.13.1 Form of Explicit Filter Elements

1090 Explicit filters can be attached to various elements in SCA, such as consumers and channels. The
1091 syntax used to express the filters conveys three things:

1092 1. The type of data that the filter operates against (the “subject”)

1093 2. The language used to express the filter (the “dialect”)

1094 3. The filter expression itself.

1095 The choice of dialect might be constrained by the choice of subject; there are some
1096 dialect/subject combinations that do not make sense.

1097 The filters, if any, that are attached to a consumer of channel are all contained in a single
1098 <sca:filters> element. The filters themselves MUST appear as child element of <sca:filters> and
1099 any element that is included as a child element of <sca:filters> MUST be a filter. The QName of
1100 the element indicates the subject of the filter and its dialect; SCA provides element declarations
1101 for all the filter subjects that it defines.

The element content is used to convey the expression, and is constrained by the dialect chosen.

The SCA specification defines a number of predefined filter subject/dialect elements. These are described in the following sections, but are summarized in this pseudo-schema:

```
<filters>
   <type qnames="list of xs:QName"? namespaces="list of xs:anyURI"?  />*
   <body.xpath1> xs:string </body.xpath1>*


   <metadata.xpath1> xs:string </metadata.xpath1>*


   <any/>*
</filters> ?
```

Note that the event filters are extensible, allowing new filter types to be defined as an extension and to be used at the place that the <any/> subelement is shown in the pseudo-schema. An SCA runtime is not required to support filter types not defined by this specification - but if an extended filter type is declared within a <filters/> element and the SCA runtime does not support that extended filter type, then the SCA runtime MUST generate an error when it encounters the declaration.

### 1.13.2 Event Type Filters

Event type filters filter events based on the Event Type metadata of the event.

Only one dialect is currently defined for event type with the element name <sca:type>

```
<filters>
   <type qnames="list of xs:QName"? namespaces="list of xs:anyURI"? />*
   ...
</filters>
```

In this dialect, a filter expression consists of either a list of one or more QNames specified as a value of the attribute @qname or a list of one of more namespace URIs specified as a value of the attribute @namespaces or both.

Each QName in the list MUST be associated with a Namespace URI. This association is performed using the namespace declarations that are in-scope where the QName expression appears (e.g. in the SCDL document containing sca:Filter element). Unprefixed QNames are permitted, provided there is a default namespace declaration in-scope where the QName expression appears. QNames that belong to no namespace are not allowed.

A filter expressed in this dialect returns true if and only if either of the following is true:

1. at least one of the QNames specified in the @qnames attribute matches the QName of the event's Event Type. In order for a match to occur both these conditions must be true:
    - The associated Namespace URI's must contain an identical sequence of characters when expressed as Unicode code points.
    - The local parts of each QName must contain an identical sequence of characters when expressed as Unicode code points.
2. at least one of the namespaces specified in the @namespace attribute matches the namespace of the event's Event Type.

- The Namespace URI's must contain an identical sequence of characters when expressed as Unicode code points.

#### 1.13.2.1 Event Type Filter Examples

A filter that expresses interest in the events of types ns1:printer or ns2:printer:

```
<type qnames="ns1:printer ns2:printer" />
```

A filter that expresses interest in events that do not have a type metadata:

```
<type qnames="sca:NULL" />
```

A filter that expresses interest in events that either do not have type metadata or are of type ns1:printer:

```
<type qnames="sca:NULL ns1:printer" />
```

A filter that expresses interest in events whose type belongs to one of the two namespaces "http://example.org/ns1" or "http://example.org/ns2":

```
<type namespaces="http://example.org/ns1 http://example.org/ns2" />
```

A filter that expresses interest in events whose type belongs to the namespaces http://example.org/ns2 or is of type ns1:printer or is untyped:

```
<type qnames="ns1:printer sca:NULL"
        namespaces="http://example.org/ns2" />
```

### 1.13.3 Business Data Filters

Business data filters filter events based on the business data contained within the event.

Element name: <sca:body.*dialect*>

The following dialects are defined - xpath1.

#### 1.13.3.1 XPATH 1.0 Dialect

Filter element QName: <sca:body.xpath1>

The Filter expression is an XPath 1.0 expression (not a predicate) whose context is:

- Context Node: the root element of the document being searched based upon the subject. In this case (the Business Data Subject) it is the root element of the event business data.
- Context Position: 1
- Context Size: 1
- Variable Binding: None
- Function Libraries: Core function library
- Namespace Declarations: Any namespace declarations in-scope where the XPath expression appears (e.g. in the SCDL document containing sca:Filter element)

This XPath expression can evaluate to one of four possible types: a node-set, a boolean, a number or a string. These result types are converted to a boolean value as follows:

1188 • Node-set – false if no nodes, true otherwise
1189 • boolean – no conversion
1190 • string – false is empty string, true otherwise
1191 • number – false if 0, true otherwise
1192

### 1193 1.13.4 Event Metadata Filters

1194

1195 Event metadata filters filter events based on the content of the event metadata.

1196 Since specific event metadata is an optional feature of SCA Event Processing, it cannot be
1197 guaranteed that an event metadata filter will find any metadata on which to operate. In this
1198 case, at runtime the filter produces a "false" result and the event is rejected by the filter.

1199 Element name: <sca:metadata.*dialect*>

1200 Event metadata is modelled as an XML document in which each piece of metadata appears as an
1201 immediate child element of the document root. This document is separate from the business data
1202 portion of the event. The following dialects are defined:

1203 • XPATH 1.0 This is the same as the XPATH 1.0 dialect used for Business data
1204 except that the Context Node is the root of the event metadata

1205

# 2 Appendix 1

## 2.1 XML Schemas

### 2.1.1 sca.xsd

This section remains unchanged from the original OSOA SCA Assembly Specification V1.00 [1].

### 2.1.2 sca-core.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2006, 2007 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
     targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
     xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
     elementFormDefault="qualified">

 <element name="componentType" type="sca:ComponentType"/>
 <complexType name="ComponentType">
   <sequence>
         <element ref="sca:implementation" minOccurs="0" maxOccurs="1"/>
         <choice minOccurs="0" maxOccurs="unbounded">
               <element name="service" type="sca:ComponentService" />
               <element name="reference" type="sca:ComponentReference"/>
               <element name="property" type="sca:Property"/>
               <element name="consumer" type="sca:ComponentConsumer"/>
               <element name="producer" type="sca:ComponentProducer"/>
         </choice>
         <any namespace="##other" processContents="lax" minOccurs="0"
               maxOccurs="unbounded"/>
     </sequence>
     <attribute name="constrainingType" type="QName" use="optional"/>
     <anyAttribute namespace="##any" processContents="lax"/>
 </complexType>

 <element name="composite" type="sca:Composite"/>
 <complexType name="Composite">
      <sequence>
          <element name="include" type="anyURI" minOccurs="0"
               maxOccurs="unbounded"/>
          <choice minOccurs="0" maxOccurs="unbounded">
              <element name="service" type="sca:Service"/>
              <element name="property" type="sca:Property"/>
              <element name="component" type="sca:Component"/>
              <element name="reference" type="sca:Reference"/>
              <element name="wire" type="sca:Wire"/>
              <element name="consumer" type="sca:Consumer"/>
              <element name="producer" type="sca:Producer"/>
              <element name="channel" type="sca:Channel"/>
          </choice>
          <any namespace="##other" processContents="lax" minOccurs="0"
```

```
1258                    maxOccurs="unbounded"/>
1259          </sequence>
1260          <attribute name="name" type="NCName" use="required"/>
1261          <attribute name="targetNamespace" type="anyURI" use="required"/>
1262          <attribute name="local" type="boolean" use="optional" default="false"/>
1263          <attribute name="autowire" type="boolean" use="optional"
1264                           default="false"/>
1265          <attribute name="constrainingType" type="QName" use="optional"/>
1266          <attribute name="requires" type="sca:listOfQNames" use="optional"/>
1267          <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
1268          <anyAttribute namespace="##any" processContents="lax"/>
1269      </complexType>
1270
1271      <complexType name="Service">
1272        <sequence>
1273              <element ref="sca:interface" minOccurs="0" maxOccurs="1" />
1274              <element name="operation" type="sca:Operation" minOccurs="0"
1275                    maxOccurs="unbounded" />
1276              <choice minOccurs="0" maxOccurs="unbounded">
1277                    <element ref="sca:binding" />
1278                    <any namespace="##other" processContents="lax"
1279                          minOccurs="0" maxOccurs="unbounded" />
1280              </choice>
1281              <element ref="sca:callback" minOccurs="0" maxOccurs="1" />
1282              <any namespace="##other" processContents="lax" minOccurs="0"
1283                    maxOccurs="unbounded" />
1284        </sequence>
1285        <attribute name="name" type="NCName" use="required" />
1286        <attribute name="promote" type="anyURI" use="required" />
1287        <attribute name="requires" type="sca:listOfQNames" use="optional" />
1288        <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
1289        <anyAttribute namespace="##any" processContents="lax" />
1290      </complexType>
1291
1292      <element name="interface" type="sca:Interface" abstract="true" />
1293      <complexType name="Interface" abstract="true"/>
1294
1295      <complexType name="Reference">
1296        <sequence>
1297              <element ref="sca:interface" minOccurs="0" maxOccurs="1" />
1298              <element name="operation" type="sca:Operation" minOccurs="0"
1299                    maxOccurs="unbounded" />
1300              <choice minOccurs="0" maxOccurs="unbounded">
1301                    <element ref="sca:binding" />
1302                    <any namespace="##other" processContents="lax" />
1303              </choice>
1304              <element ref="sca:callback" minOccurs="0" maxOccurs="1" />
1305              <any namespace="##other" processContents="lax" minOccurs="0"
1306                    maxOccurs="unbounded" />
1307        </sequence>
1308        <attribute name="name" type="NCName" use="required" />
1309        <attribute name="target" type="sca:listOfAnyURIs" use="optional"/>
1310        <attribute name="wiredByImpl" type="boolean" use="optional"
1311                           default="false"/>
1312        <attribute name="multiplicity" type="sca:Multiplicity"
1313              use="optional" default="1..1" />
1314        <attribute name="promote" type="sca:listOfAnyURIs" use="required" />
1315        <attribute name="requires" type="sca:listOfQNames" use="optional" />
1316        <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
```

```
1317        <anyAttribute namespace="##any" processContents="lax" />
1318      </complexType>
1319
1320      <complexType name="Consumer">
1321        <complexContent>
1322              <extension base="sca:ComponentConsumer">
1323                    <attribute name="promote" type="sca:listOfAnyURIs"
1324                     use="required" />
1325              </extension>
1326        </complexContent>
1327      </complexType>
1328
1329      <complexType name="Producer">
1330        <complexContent>
1331              <extension base="sca:ComponentProducer">
1332                    <attribute name="promote" type="sca:listOfAnyURIs"
1333                     use="required" />
1334              </extension>
1335        </complexContent>
1336      </complexType>
1337
1338      <complexType name="Channel">
1339        <sequence>
1340              <element ref="sca:filters" minOccurs="0" maxOccurs="1" />
1341              <element ref="sca:binding" minOccurs="0" maxOccurs="unbounded"/>
1342              <any namespace="##other" processContents="lax" minOccurs="0"
1343                    maxOccurs="unbounded" />
1344        </sequence>
1345        <attribute name="name" type="NCName" use="required" />
1346        <attribute name="requires" type="sca:listOfQNames" use="optional" />
1347        <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
1348        <anyAttribute namespace="##any" processContents="lax" />
1349      </complexType>
1350
1351      <complexType name="SCAPropertyBase" mixed="true">
1352        <!-- mixed="true" to handle simple type -->
1353        <sequence>
1354              <any namespace="##any" processContents="lax" minOccurs="0"
1355                    maxOccurs="1" />
1356              <!-- NOT an extension point; This xsd:any exists to accept
1357                     the element-based or complex type property
1358                     i.e. no element-based extension point under "sca:property"
1359              -->
1360        </sequence>
1361      </complexType>
1362
1363      <!-- complex type for sca:property declaration -->
1364      <complexType name="Property" mixed="true">
1365         <complexContent>
1366               <extension base="sca:SCAPropertyBase">
1367                  <!-- extension defines the place to hold default value -->
1368                  <attribute name="name" type="NCName" use="required"/>
1369                  <attribute name="type" type="QName" use="optional"/>
1370                  <attribute name="element" type="QName" use="optional"/>
1371                  <attribute name="many" type="boolean" default="false"
1372                    use="optional"/>
1373                  <attribute name="mustSupply" type="boolean" default="false"
1374                    use="optional"/>
1375                  <anyAttribute namespace="##any" processContents="lax"/>
```

```
1376                <!-- an extension point ; attribute-based only -->
1377             </extension>
1378       </complexContent>
1379    </complexType>
1380
1381    <complexType name="PropertyValue" mixed="true">
1382       <complexContent>
1383          <extension base="sca:SCAPropertyBase">
1384             <attribute name="name" type="NCName" use="required"/>
1385             <attribute name="type" type="QName" use="optional"/>
1386             <attribute name="element" type="QName" use="optional"/>
1387             <attribute name="many" type="boolean" default="false"
1388                  use="optional"/>
1389             <attribute name="source" type="string" use="optional"/>
1390             <attribute name="file" type="anyURI" use="optional"/>
1391             <anyAttribute namespace="##any" processContents="lax"/>
1392             <!-- an extension point ; attribute-based only -->
1393          </extension>
1394       </complexContent>
1395    </complexType>
1396
1397    <element name="binding" type="sca:Binding" abstract="true"/>
1398    <complexType name="Binding" abstract="true">
1399      <sequence>
1400            <element name="operation" type="sca:Operation" minOccurs="0"
1401                  maxOccurs="unbounded" />
1402      </sequence>
1403        <attribute name="uri" type="anyURI" use="optional"/>
1404        <attribute name="name" type="NCName" use="optional"/>
1405        <attribute name="requires" type="sca:listOfQNames" use="optional"/>
1406        <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
1407    </complexType>
1408
1409    <element name="bindingType" type="sca:BindingType"/>
1410    <complexType name="BindingType">
1411      <sequence minOccurs="0" maxOccurs="unbounded">
1412            <any namespace="##other" processContents="lax" />
1413      </sequence>
1414      <attribute name="type" type="QName" use="required"/>
1415      <attribute name="alwaysProvides" type="sca:listOfQNames" use="optional"/>
1416      <attribute name="mayProvide" type="sca:listOfQNames" use="optional"/>
1417      <anyAttribute namespace="##any" processContents="lax"/>
1418    </complexType>
1419
1420    <element name="callback" type="sca:Callback"/>
1421    <complexType name="Callback">
1422      <choice minOccurs="0" maxOccurs="unbounded">
1423            <element ref="sca:binding"/>
1424            <any namespace="##other" processContents="lax"/>
1425        </choice>
1426        <attribute name="requires" type="sca:listOfQNames" use="optional"/>
1427        <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
1428        <anyAttribute namespace="##any" processContents="lax"/>
1429    </complexType>
1430
1431    <complexType name="Component">
1432        <sequence>
1433             <element ref="sca:implementation" minOccurs="0" maxOccurs="1"/>
1434             <choice minOccurs="0" maxOccurs="unbounded">
```

```
1435                <element name="service" type="sca:ComponentService"/>
1436                <element name="reference" type="sca:ComponentReference"/>
1437                <element name="property" type="sca:PropertyValue" />
1438                <element name="consumer" type="sca:ComponentConsumer" />
1439                <element name="producer" type="sca:ComponentProducer" />
1440             </choice>
1441             <any namespace="##other" processContents="lax" minOccurs="0"
1442                  maxOccurs="unbounded"/>
1443         </sequence>
1444         <attribute name="name" type="NCName" use="required"/>
1445         <attribute name="autowire" type="boolean" use="optional"
1446                         default="false"/>
1447         <attribute name="constrainingType" type="QName" use="optional"/>
1448         <attribute name="requires" type="sca:listOfQNames" use="optional"/>
1449         <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
1450         <anyAttribute namespace="##any" processContents="lax"/>
1451     </complexType>
1452
1453     <complexType name="ComponentService">
1454       <complexContent>
1455             <restriction base="sca:Service">
1456                     <sequence>
1457                          <element ref="sca:interface" minOccurs="0"
1458                                maxOccurs="1"/>
1459                          <element name="operation" type="sca:Operation"
1460                                minOccurs="0" maxOccurs="unbounded" />
1461                          <choice minOccurs="0" maxOccurs="unbounded">
1462                             <element ref="sca:binding"/>
1463                               <any namespace="##other" processContents="lax"
1464                                     minOccurs="0" maxOccurs="unbounded"/>
1465                          </choice>
1466                          <element ref="sca:callback" minOccurs="0"
1467                                maxOccurs="1"/>
1468                          <any namespace="##other" processContents="lax"
1469                                minOccurs="0" maxOccurs="unbounded"/>
1470                     </sequence>
1471                     <attribute name="name" type="NCName" use="required"/>
1472                     <attribute name="requires" type="sca:listOfQNames"
1473                         use="optional"/>
1474                     <attribute name="policySets" type="sca:listOfQNames"
1475                         use="optional"/>
1476                     <anyAttribute namespace="##any" processContents="lax"/>
1477             </restriction>
1478       </complexContent>
1479     </complexType>
1480
1481     <complexType name="ComponentReference">
1482       <complexContent>
1483             <restriction base="sca:Reference">
1484                   <sequence>
1485                         <element ref="sca:interface" minOccurs="0" maxOccurs="1"
1486                          />
1487                         <element name="operation" type="sca:Operation"
1488                                      minOccurs="0" maxOccurs="unbounded" />
1489                         <choice minOccurs="0" maxOccurs="unbounded">
1490                               <element ref="sca:binding" />
1491                               <any namespace="##other" processContents="lax" />
1492                         </choice>
1493                         <element ref="sca:callback" minOccurs="0" maxOccurs="1"
```

```
1494                       />
1495                     <any namespace="##other" processContents="lax"
1496                            minOccurs="0" maxOccurs="unbounded" />
1497               </sequence>
1498               <attribute name="name" type="NCName" use="required" />
1499               <attribute name="autowire" type="boolean" use="optional"
1500                     default="false"/>
1501               <attribute name="wiredByImpl" type="boolean" use="optional"
1502                     default="false"/>
1503               <attribute name="target" type="sca:listOfAnyURIs"
1504                            use="optional"/>
1505               <attribute name="multiplicity" type="sca:Multiplicity"
1506                     use="optional" default="1..1" />
1507               <attribute name="requires" type="sca:listOfQNames"
1508                            use="optional"/>
1509               <attribute name="policySets" type="sca:listOfQNames"
1510                     use="optional"/>
1511               <anyAttribute namespace="##any" processContents="lax" />
1512         </restriction>
1513      </complexContent>
1514    </complexType>
1515
1516   <complexType name="ComponentConsumer">
1517      <sequence>
1518            <element ref="sca:filters" minOccurs="0" maxOccurs="1" />
1519            <element ref="sca:binding" minOccurs="0" maxOccurs="unbounded"/>
1520            <any namespace="##other" processContents="lax" minOccurs="0"
1521                  maxOccurs="unbounded" /><
1522      </sequence>
1523      <attribute name="name" type="NCName" use="required" />
1524      <attribute name="requires" type="sca:listOfQNames" use="optional" />
1525      <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
1526      <anyAttribute namespace="##any" processContents="lax" />
1527    </complexType>
1528
1529    <complexType name="ComponentProducer">
1530      <sequence>
1531            <element ref="sca:binding" minOccurs="0" maxOccurs="unbounded"/>
1532            <any namespace="##other" processContents="lax" minOccurs="0"
1533                  maxOccurs="unbounded" /><
1534      </sequence>
1535      <attribute name="name" type="NCName" use="required" />
1536      <attribute name="requires" type="sca:listOfQNames" use="optional" />
1537      <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
1538      <attribute name="typeNames" type="sca:listOfQNames" use="optional"/>
1539      <attribute name="typeNamespaces" type="sca:listOfAnyURIs" use="optional"/>
1540      <anyAttribute namespace="##any" processContents="lax" />
1541    </complexType>
1542
1543    <element name="implementation" type="sca:Implementation"
1544      abstract="true" />
1545    <complexType name="Implementation" abstract="true">
1546      <attribute name="requires" type="sca:listOfQNames" use="optional"/>
1547      <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
1548    </complexType>
1549
1550    <element name="implementationType" type="sca:ImplementationType"/>
1551    <complexType name="ImplementationType">
1552      <sequence minOccurs="0" maxOccurs="unbounded">
```

```
1553                <any namespace="##other" processContents="lax" />
1554          </sequence>
1555          <attribute name="type" type="QName" use="required"/>
1556          <attribute name="alwaysProvides" type="sca:listOfQNames" use="optional"/>
1557          <attribute name="mayProvide" type="sca:listOfQNames" use="optional"/>
1558          <anyAttribute namespace="##any" processContents="lax"/>
1559      </complexType>
1560
1561      <complexType name="Wire">
1562          <sequence>
1563              <any namespace="##other" processContents="lax" minOccurs="0"
1564                    maxOccurs="unbounded"/>
1565          </sequence>
1566          <attribute name="source" type="anyURI" use="required"/>
1567          <attribute name="target" type="anyURI" use="required"/>
1568          <anyAttribute namespace="##any" processContents="lax"/>
1569      </complexType>
1570
1571      <element name="include" type="sca:Include"/>
1572      <complexType name="Include">
1573              <attribute name="name" type="QName"/>
1574              <anyAttribute namespace="##any" processContents="lax"/>
1575        </complexType>
1576
1577      <complexType name="Operation">
1578        <attribute name="name" type="NCName" use="required"/>
1579        <attribute name="requires" type="sca:listOfQNames" use="optional"/>
1580        <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
1581        <anyAttribute namespace="##any" processContents="lax"/>
1582      </complexType>
1583
1584      <element name="constrainingType" type="sca:ConstrainingType"/>
1585      <complexType name="ConstrainingType">
1586          <sequence>
1587              <choice minOccurs="0" maxOccurs="unbounded">
1588                  <element name="service" type="sca:ComponentService"/>
1589                  <element name="reference" type="sca:ComponentReference"/>
1590                  <element name="property" type="sca:Property" />
1591              </choice>
1592              <any namespace="##other" processContents="lax" minOccurs="0"
1593                    maxOccurs="unbounded"/>
1594          </sequence>
1595          <attribute name="name" type="NCName" use="required"/>
1596          <attribute name="targetNamespace" type="anyURI"/>
1597          <attribute name="requires" type="sca:listOfQNames" use="optional"/>
1598          <anyAttribute namespace="##any" processContents="lax"/>
1599      </complexType>
1600
1601
1602      <simpleType name="Multiplicity">
1603          <restriction base="string">
1604              <enumeration value="0..1"/>
1605              <enumeration value="1..1"/>
1606              <enumeration value="0..n"/>
1607              <enumeration value="1..n"/>
1608          </restriction>
1609      </simpleType>
1610
1611      <simpleType name="OverrideOptions">
```

```
1612          <restriction base="string">
1613              <enumeration value="no"/>
1614              <enumeration value="may"/>
1615              <enumeration value="must"/>
1616          </restriction>
1617      </simpleType>
1618
1619      <!-- Global attribute definition for @requires to permit use of intents
1620           within WSDL documents -->
1621      <attribute name="requires" type="sca:listOfQNames"/>
1622
1623      <!-- Global attribute defintion for @endsConversation to mark operations
1624           as ending a conversation -->
1625      <attribute name="endsConversation" type="boolean" default="false"/>
1626
1627      <simpleType name="listOfQNames">
1628        <list itemType="QName"/>
1629      </simpleType>
1630
1631      <simpleType name="listOfAnyURIs">
1632          <list itemType="anyURI"/>
1633      </simpleType>
1634
1635      <element name="filters" type="sca:Filter"/>
1636      <complexType name="Filter">
1637          <sequence>
1638              <choice minOccurs="0" maxOccurs="unbounded">
1639                    <element ref="sca:type" />
1640                    <element ref="sca:body.xpath1" />
1641                    <element ref="sca:metadata.xpath1" />
1642              </choice>
1643              <any namespace="##other" processContents="lax" minOccurs="0"
1644                    maxOccurs="unbounded"/>
1645          </sequence>
1646          <anyAttribute namespace="##other" processContents="lax"/>
1647      </complexType>
1648
1649      <element name="type" type="sca:Type"/>
1650      <complexType name="Type">
1651          <sequence>
1652              <any namespace="##other" processContents="lax" minOccurs="0"
1653                    maxOccurs="unbounded"/>
1654          </sequence>
1655          <attribute name="qnames" type="sca:listOfQNames" />
1656          <attribute name="namespaces" type="sca:listOfAnyURIs" />
1657          <anyAttribute namespace="##other" processContents="lax" />
1658      </complexType>
1659
1660      <element name="body.xpath1" type="string" />
1661
1662      <element name="metadata.xpath1" type="string" />
1663
1664  </schema>
1665
```

1666 ### 2.1.3 sca-binding-sca.xsd

1667

1668 This section remains unchanged from the original OSOA SCA Assembly Specification V1.00 [1].

1669

### 2.1.4 sca-interface-java.xsd

This section remains unchanged from the original OSOA SCA Assembly Specification V1.00 [1].

### 2.1.5 sca-interface-wsdl.xsd

This section remains unchanged from the original OSOA SCA Assembly Specification V1.00 [1].

### 2.1.6 sca-implementation-java.xsd

This section remains unchanged from the original OSOA SCA Assembly Specification V1.00 [1].

### 2.1.7 sca-implementation-composite.xsd

This section remains unchanged from the original OSOA SCA Assembly Specification V1.00 [1].

### 2.1.8 sca-definitions.xsd

This section remains unchanged from the original OSOA SCA Assembly Specification V1.00 [1].

### 2.1.9 sca-binding-webservice.xsd

Is described in the SCA Web Services Binding specification [9]

### 2.1.10 sca-binding-jms.xsd

Is described in the SCA JMS Binding specification [11]

### 2.1.11 sca-policy.xsd

Is described in the SCA Policy Framework specification [10]

### 2.1.12 sca-eventDefinition.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2009 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
    xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
    elementFormDefault="qualified">

```

```
1711        <element name="eventDefinitions" type="sca:EventDefinitions" />
1712        <complexType name="EventDefinitions" >
1713           <sequence>
1714                 <element ref="sca:eventType" minOccurs="0" maxOccurs="unbounded"
1715                  />
1716                 <any namespace="##other" processContents="lax" minOccurs="0"
1717                  maxOccurs="unbounded"/>
1718           </sequence>
1719           <attribute name="targetNamespace" type="anyURI" use="required" />
1720           <anyAttribute namespace="##other" processContents="lax" />
1721        </complexType>
1722
1723        <element name="eventType" type="sca:EventType" abstract="true" />
1724        <complexType name="EventType" abstract="true" >
1725           <attribute name="name" type="NCName" />
1726        </complexType>
1727
1728        <element name="eventType.xsd" type="sca:XSDEventType"
1729            substitutionGroup="sca:eventType" />
1730        <complexType name="XSDEventType" >
1731           <complexContent>
1732             <extension base="sca:EventType">
1733                <attribute name="element" type="QName" use="required"/>
1734                <anyAttribute namespace="##other" processContents="lax"/>
1735             </extension>
1736           </complexContent>
1737        </complexType>
1738
1739    </schema>
1740
```

1741 ## *2.2 SCA Concepts*
1742

1743 This section remains unchanged from the original OSOA SCA Assembly Specification V1.00 [1].
1744

# 3 Java Implementation Type

This appendix describes an outline of how SCA Event Processing capabilities are handled for implementations that are written in the Java language. This section is not normative and is provided for information only. Once the SCA Assembly extensions for Event Processing are accepted, it is anticipated that the SCA Java Implementation specifications will be modified and extended to describe these capabilities normatively.

The Java Implementation for Event Processing is an extension of the standard SCA Java Implementation model. All the standard SCA Java implementation features continue to be available to Java implementations. The following features are added:

- Ability to define one or more methods of the implementation class to be ***consumer methods***, consuming one or more ***event types***
- Ability to define a field or setter method of the implementation class as an event ***producer***, with one or more business methods producing one or more ***event types***
- Ability to define ***event types*** as Java POJO classes

## 3.1 Event Consumer methods

Where a Java implementation needs to receive events and process them, it declares one or more methods of the implementation as ***event consumer*** methods.

Each method of the implementation that is a consumer for events is annotated with a ***@Consumer*** annotation. Each method with a @Consumer annotation must have a ***void*** return type and a single parameter that is either a specific event type or a superclass of one or more event types, including java.lang.Object, which is treated as the supertype of all event types. The @Consumer annotation has the ***consumer name*** as a required parameter.

Where a consumer method handles more than one event type, the set of event types accepted by the method can optionally be declared using the ***@EventTypes*** annotation on the method. The @EventTypes annotation contains a list of one or more event type names.

Multiple methods of the implementation can be annotated with the @Consumer annotation. Where the @Consumer annotations have different consumer names, each method is treated as a separate consumer, receiving different streams of events. Where two or more @Consumer annotations have the same consumer name, all the methods with the same consumer name are handled as part of a single consumer, sharing a single event stream. Where multiple methods are declared to belong to the same consumer, each method MUST deal with a separate set of event types - any event received by the implementation is only sent to ***one*** of the methods.

Each consumer forms part of the Java implementation's component type and is configured to receive events from (zero or more) event producers by the component which uses the implementation.

The following example is a Java class with a single method ***someBusinessMethod*** which is annotated with @Consumer, with the consumer name set to ***ExampleConsumer***. In this case the event type accepted by the method is defined by the parameter of the method being a class that is annotated with the @EventType annotation.

```
public class ConsumerExample {

```

```
1789        @Consumer(ExampleConsumer)
1790        public void someBusinessMethod( ExampleEvent theEvent ) {
1791            String businessData = theEvent.eventData;
1792            // do business processing...
1793        } // end method someBusinessMethod
1794
1795    } // end class
1796
```

1797 ## 3.2 Event Producers

1798

1799 Where a Java implementation needs to produce and send events for others to consumer, it
1800 declares an ***event producer***, which is identified as a Field or a Setter method annotated with a
1801 ***@Producer*** annotation.

1802 It is required that the Field or Setter method annotated with @Producer is typed by a Java
1803 interface. The Java interface must have one or more methods, each of which has a ***void*** return
1804 type and a single parameter that is either a specific event type or a superclass of one or more
1805 event types, including java.lang.Object, which is treated as the supertype of all event types.

1806 Where a producer method handles more than one event type, the set of event types produced by
1807 the method can be declared using the ***@EventTypes*** annotation on the method. The
1808 @EventTypes annotation contains a list of one or more event type names. This is particularly
1809 useful if the formal method parameter is some general type such as java.lang.Object.

1810 The @Producer annotation has the ***producer name*** as a required parameter. Multiple fields
1811 and/or setter methods can be annotated with the @Producer annotation, but each one MUST use
1812 a unique producer name. Each distinct producer represents a different stream of events. Each
1813 producer forms part of the component type of the Java implementation and is configured to send
1814 events to (zero or more) event consumers by the component which uses the implementation.

1815 The following example is a Java class with a setter method called ***setEventProducer*** which is
1816 annotated with @Producer and which is given the producer name ***ExampleEvent***. The event
1817 type handled by this producer is declared as the parameter of the ***produceExampleEvent***
1818 method in the ***ExampleProducer*** interface.

1819

```
1820    @Remotable
1821    public interface ExampleProducer {
1822
1823        void produceExampleEvent( ExampleEvent theEvent);
1824
1825    } // end interface ExampleProducer
1826
1827    @EventType(ExampleEvent)
1828
1829    public class ExampleEvent {
1830
1831         public String eventData;
1832
1833    } // end class ExampleEvent
1834
1835    public class ProducerExample {
```

```
    private ExampleProducer eventProducer;

    @Producer(ExampleEvent)
    public void setEventProducer( ExampleProducer theProducer ) {
        eventProducer = theProducer;
        return;
    } // end method setEventProducer

    public void someBusinessMethod() {
        theEvent = new ExampleEvent();
        theEvent.eventData = "Some Data";
        eventProducer.produceExampleEvent( theEvent );
    } // end method someBusinessMethod

} // end class
```

## 3.3 Event Types

Event types are defined through Java classes which are annotated with a ***@EventType*** annotation. The @EventType annotation has a single parameter which is the ***name*** of the event type.

```
@EventType(ExampleEventType)
public class ExampleEvent {


     public String eventData;


} // end class ExampleEvent
```

The name may be:

- ***unqualified***, in which case the EventType name is qualified by the package name of the Java class itself
- ***qualified***, in which case the EventType name is used as it is declared

Note that the Event Type name maps to an XSD QName using the Java-to-WSDL mapping as defined by JAX-WS.

The Event type(s) handled by a consumer method or a producer method is defined in one of two ways:

1. The method has a parameter that is of a class annotated with the @EventType annotation (of a single event type)
2. The method is itself annotated with the @EventTypes annotation for one or more event types - where the method parameter is of some generic type such as java.lang.Object which does not directly specify an EventType

# 4 References

[1] SCA Assembly Model Specification
http://www.osoa.org/download/attachments/35/SCA_AssemblyModel_V100.pdf

[2] SCA Assembly Model Specification Version 1.1
http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec.pdf

[3] SCA Example Code document
http://www.osoa.org/download/attachments/28/SCA_BuildingYourFirstApplication_V09.pdf

[4] JAX-WS Specification
http://jcp.org/en/jsr/detail?id=101

[5] WS-I Basic Profile
http://www.ws-i.org/deliverables/workinggroup.aspx?wg=basicprofile

[6] WS-I Basic Security Profile
http://www.ws-i.org/deliverables/workinggroup.aspx?wg=basicsecurity

[7] Business Process Execution Language (BPEL)
http://www.oasis-open.org/committees/documents.php?wg_abbrev=wsbpel

[8] WSDL Specification
WSDL 1.1: http://www.w3.org/TR/wsdl
WSDL 2.0: http://www.w3.org/TR/wsdl20/

[9] SCA Web Services Binding Specification
http://www.osoa.org/download/attachments/35/SCA_WebServiceBindings_V100.pdf

[10] SCA Policy Framework Specification
http://www.osoa.org/download/attachments/35/SCA_Policy_Framework_V100.pdf

[11] SCA JMS Binding Specification
http://www.osoa.org/download/attachments/35/SCA_JMSBinding_V100.pdf

[12] ZIP Format Definition
http://www.pkware.com/documents/casestudies/APPNOTE.TXT