



Service Component Architecture Client and Implementation Model Specification for C++ Version 1.1

Committee Draft 01

20 March 2008

Specification URIs:

This Version:

<http://docs.oasis-open.org/opencsa/sca-c-cpp/sca-cppcni-1.1-spec-cd-01.html>
<http://docs.oasis-open.org/opencsa/sca-c-cpp/sca-cppcni-1.1-spec-cd-01.doc>
<http://docs.oasis-open.org/opencsa/sca-c-cpp/sca-cppcni-1.1-spec-cd-01.pdf> (Authoritative)

Previous Version:

N/A

Latest Version:

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

Latest Approved Version:

Technical Committee:

OASIS Service Component Architecture / C and C++ (SCA-C-C++) TC

Chair:

Bryan Aupperle, IBM <<mailto:aupperle@us.ibm.com>>

Editors:

Bryan Aupperle, IBM <<mailto:aupperle@us.ibm.com>>
David Haney, Rogue Wave Software <<mailto:haney@roguewave.com>>
Pete Robbins, IBM <<mailto:robbins@uk.ibm.com>>

Related work:

This specification replaces or supercedes:

- [OSOA SCA C++ Client and Implementation V1.00](#)

This specification is related to:

- [OASIS Service Component Architecture Assembly Model Version 1.1](#)
- [OASIS SCA Policy Framework Version 1.1](#)
- [OASIS Service Component Architecture Web Service Binding Specification Version 1.1](#)

Declared XML Namespace(s):

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

Abstract:

This document describes the SCA Client and Implementation Model for the C++ programming language.

The SCA C++ implementation model describes how to implement SCA components in C++. A component implementation itself can also be a client to other services provided by other components or external services. The document describes how a C++ implemented component gets access to services and calls their methods.

The document also explains how non-SCA C++ components can be clients to services provided by other components or external services. The document shows how those non-SCA C++ component implementations access services and call their methods.

Status:

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

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

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

Notices

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

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

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

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

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

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

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

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

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

Table of Contents

1	Introduction	7
1.1	Terminology	7
1.2	Normative References	7
1.3	Non-Normative References	7
2	Basic Component Implementation Model	8
2.1	Implementing a Service	8
2.1.1	Implementing a Remotable Service	9
2.1.2	Implementing a Local Service	10
2.2	Conversational and Non-Conversational services	10
2.3	Component Implementation Scopes	10
2.3.1	Stateless scope	11
2.3.2	Request scope	11
2.3.3	Composite scope	11
2.3.4	Conversation scope	11
2.4	Implementing a Configuration Property	12
2.5	Component Type and Component	12
2.5.1	Interface.cpp	13
2.5.2	Method	14
2.5.3	Implementation.cpp	14
2.5.4	Implementation Method	15
2.6	Instantiation	16
3	Basic Client Model	17
3.1	Accessing Services from Component Implementations	17
3.2	Accessing Services from non-SCA component implementations	18
3.3	Calling Service Methods	18
4	Error Handling	19
5	Conversational Services	20
5.1	Conversational Client	20
5.2	Conversational Service Provider	20
5.3	Methods that End the Conversation	21
5.4	Passing Conversational Services as Parameters	22
5.5	Conversation Lifetime Summary	22
5.6	Application Specified Conversation IDs	23
5.7	Accessing Conversation IDs from Clients	23
6	Asynchronous Programming	24
6.1	Non-blocking Calls	24
6.2	Callbacks	24
6.2.1	Stateful Callbacks	25
6.2.2	Stateless Callbacks	26
6.2.3	Implementing Multiple Bidirectional Interfaces	26
6.2.4	Customizing the Callback Identity	26
7	C++ API	27
7.1	Reference Counting Pointers	27

7.2	Component Context	27
7.3	ServiceReference	28
7.4	SCAException	29
7.5	SCANullPointerException	29
7.6	ServiceRuntimeException	30
7.7	ServiceUnavailableException	30
7.8	NoRegisteredCallbackException	30
7.9	ConversationEndedException	30
7.10	MultipleServicesException	31
8	WSDL to C++ and C++ to WSDL Mapping	32
9	Packaging	33
9.1	Composite Packaging	33
10	Types Supported in Service Interfaces	34
10.1	Local service	34
10.2	Remotable service	34
11	Restrictions on C++ header files	35
12	C++ to WSDL Mapping	36
12.1	Parameter and Return Type mappings	36
12.1.1	Built-in, STL and SDO type mappings	36
12.1.2	Binary data mapping	37
12.1.3	Array mapping	37
12.1.4	Multi-dimensional array mapping	38
12.1.5	Pointer/reference mapping	38
12.1.6	STL container mapping	39
12.1.7	Struct mapping	39
12.1.8	Enum mapping	40
12.1.9	Union mapping	41
12.1.10	Typedef mapping	41
12.1.11	Pre-processor mapping	41
12.1.12	Nesting types	41
12.1.13	SDO mapping	42
12.1.14	void* mapping	43
12.1.15	User-defined types (UDT) mapping	43
12.1.16	Included or Inherited types	44
12.2	Namespace mapping	44
12.3	Class mapping	44
12.4	Method mapping	46
12.4.1	Default parameter value mapping	46
12.4.2	Non-named parameters and the return type	47
12.4.3	The void return type	47
12.4.4	No Parameters Specified	49
12.4.5	In/Out Parameters	49
12.4.6	Public Methods	49
12.4.7	Inherited Public Methods	49
12.4.8	Protected/Private Methods	49

12.4.9	Constructors/Destructors.....	49
12.4.10	Overloaded Methods.....	49
12.4.11	Operator overloading.....	49
12.4.12	Exceptions.....	49
13	Conformance.....	50
A	C++ Annotations.....	51
A.1	Application of Annotations to C++ Program Elements.....	51
A.2	Interface Header Annotations.....	51
A.2.1	@Interface.....	51
A.2.2	@Remotable.....	52
A.2.3	@Callback.....	52
A.2.4	@OneWay.....	53
A.2.5	@Conversational.....	54
A.2.6	@EndsConversation.....	54
A.3	Implementation Header Annotations.....	55
A.3.1	@ComponentType.....	55
A.3.2	@Scope.....	55
A.3.3	@EagerInit.....	56
A.3.4	@AllowsPassByReference.....	56
A.3.5	@ConversationAttributes.....	57
A.3.6	@Property.....	58
A.3.7	@Reference.....	58
B	Policy Annotations for C++.....	60
B.1	General Intent Annotations.....	60
B.2	Specific Intent Annotations.....	61
B.3	Application of Intent Annotations.....	62
B.4	Inheritance and Intent Annotations.....	62
B.5	Relationship of Declarative and Annotated Intents.....	64
B.6	Policy Set Annotations.....	64
B.7	Security Policy Annotations.....	65
B.7.1	Security Interaction Policy.....	65
B.7.2	Security Implementation Policy.....	66
C	XML Schemas.....	69
C.1	sca-interface-cpp-1.1-schema.xsd.....	69
C.2	sca-implementation-cpp-1.1-schema.xsd.....	69
D	Migration.....	71
D.1	Annotations related to conversations.....	71
E	Acknowledgements.....	72
F	Non-Normative Text.....	73
G	Revision History.....	74

1 Introduction

This document describes the SCA Client and Implementation Model for the C++ programming language.

The SCA C++ implementation model describes how to implement SCA components in C++. A component implementation itself can also be a client to other services provided by other components or external services. The document describes how a C++ implemented component gets access to services and calls their methods.

The document also explains how non-SCA C++ components can be clients to services provided by other components or external services. The document shows how those non-SCA C++ component implementations access services and call their methods.

1.1 Terminology

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

1.2 Normative References

- [RFC2119] S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, <http://www.ietf.org/rfc/rfc2119.txt>, IETF RFC 2119, March 1997.
- [ASSEMBLY] M. Beisiegel, et al., *Service Component Architecture Assembly Model Specification Version 1.1*, <http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec.pdf>, OASIS Service Component Architecture Assembly Model Specification Version 1.1, XXX 2008
- [POLICY] J. Anderson, et al., *SCA Policy Framework Version 1.1*, <http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec.pdf>, OASIS SCA Policy Framework Version 1.1, XXX 2008
- [SDO21] J. Beatty, et al., *Service Data Objects For C++ Specification*, <http://www.osoa.org/download/attachments/36/Cpp-SDO-Spec-v2.1.0-FINAL.pdf>, SDO 2.1, December 2006.
- [WSDL11] E. Christensen, et al., *Web Service Description Language (WSDL)*, <http://www.w3.org/TR/wSDL>, W3C Note Web Service Description Language (WSDL), March 2001
- [WSDL20] R. Chinnici, et al., *Web Service Description Language (WSDL) Version 2.0 Part 1: Core Language*, <http://www.w3.org/TR/wSDL20/>, W3C Web Service Description Language (WSDL) Version 2.0 Part 1: Core Language, June 2007

1.3 Non-Normative References

- [CPPWSDL] *WSDL to C++ Mapping Specification*, <http://www.omg.org/docs/ptc/06-08-01.pdf>, OMG WSDL to C++ Mapping Specification, August 2006

2 Basic Component Implementation Model

38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83

This section describes how SCA components are implemented using the C++ programming language. It shows how a C++ implementation based component can implement a local or remotable service, and how the implementation can be made configurable through properties.

A component implementation can itself be a client of services. This aspect of a component implementation is described in the basic client model section.

2.1 Implementing a Service

A component implementation based on a C++ class (a *C++ implementation*) provides one or more services.

The services provided by the C++ implementation have an interface which is defined using one of:

- a C++ abstract base class
- a WSDL 1.1 portType [**WSDL11**]
- a WSDL 2.0 interface [**WSDL20**]

An abstract base class is a class which has only pure virtual methods. This is the service interface. The C++ class based component implementation **MUST** implement all of the operations of the service interface.

The following snippets show the C++ service interface and the C++ implementation class of a C++ implementation.

Service interface.

```
// LoanService interface
class LoanService {
public:
    virtual bool approveLoan(unsigned long customerNumber,
        unsigned long loanAmount) = 0;
};
```

Implementation declaration header file.

```
class LoanServiceImpl : public LoanService
{
public:
    LoanServiceImpl();
    virtual ~LoanServiceImpl();

    virtual bool approveLoan(unsigned long customerNumber,
        unsigned long loanAmount);
};
```

Implementation.


```

84 #include "LoanServiceImpl.h"
85
86 LoanServiceImpl::LoanServiceImpl ()
87 {
88     ...
89 }
90
91 LoanServiceImpl::~LoanServiceImpl ()
92 {
93     ...
94 }
95
96 bool LoanServiceImpl::approveLoan(unsigned long customerNumber,
97     unsigned long loanAmount)
98 {
99     ...
100 }

```

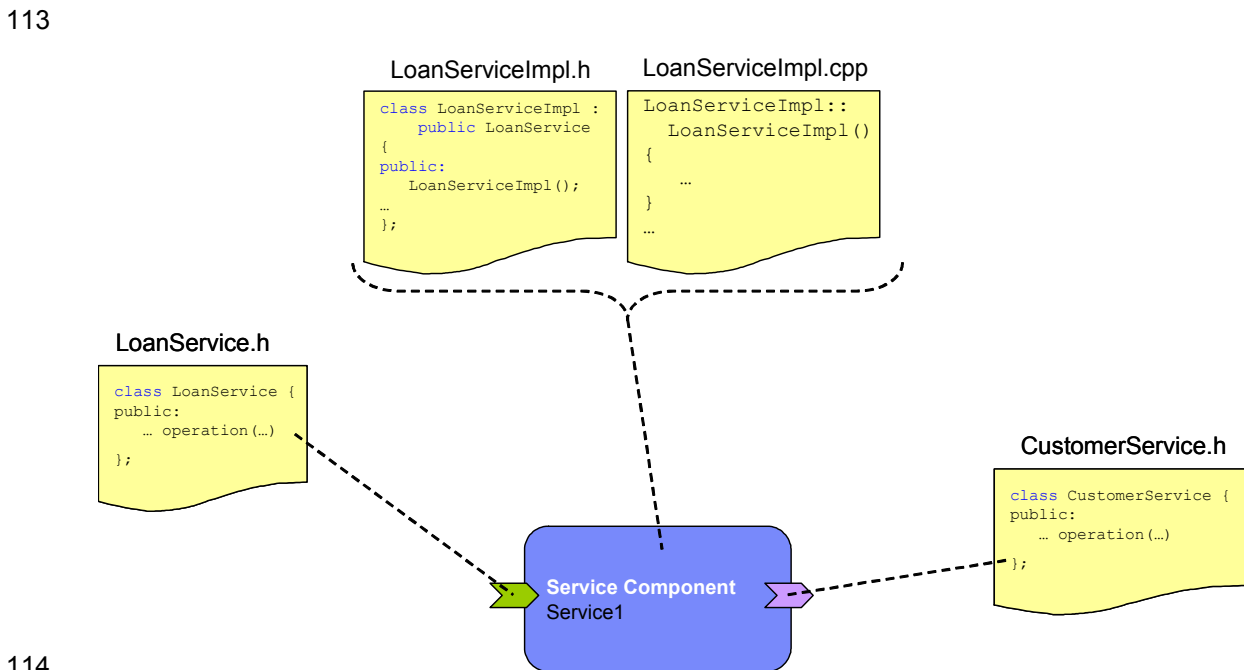
101
102 The following snippet shows the component type for this component implementation.

```

103
104 <?xml version="1.0" encoding="ASCII"?>
105 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
106     <service name="LoanService">
107         <interface.cpp header="LoanService.h"/>
108     </service>
109 </componentType>

```

110
111 The following picture shows the relationship between the C++ header files and implementation files for a
112 component that has a single service and a single reference.



114
115 **2.1.1 Implementing a Remotable Service**

116 A *remotable="true"* attribute on an interface.cpp element indicates that the interface remotable as
117 described in the Assembly Specification [ASSEMBLY]. The following snippet shows the component type
118 for a remotable service:

119

120

121

122

123

124

125

```
<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
  <service name="LoanService">
    <interface.cpp header="LoanService.h" remotable="true"/>
  </service>
</componentType>
```

126

127 Complex data types exchanged via remotable service interfaces MUST be compatible with the
128 marshalling technology that is used by the service binding.

129

130 An implementation of a remotable service can declare whether it allows pass by reference data exchange
131 semantics on calls to it, meaning that the by-value semantics can be maintained without requiring that the
132 parameters be copied. The implementation of remotable services that allow pass by reference MUST
133 NOT alter its input data during or after the invocation, and MUST NOT modify return data after invocation.
134 The *allowsPassByReference=true* attribute on the implementation.cpp element of a remotable service is
135 used to declare that calls to the whole interface allows pass by reference. Alternatively, this attribute can
136 be used on a specific method.

137 2.1.2 Implementing a Local Service

138 A service interface not marked as remotable is *local*.

139 2.2 Conversational and Non-Conversational services

140 A *requires="conversational"* attribute on an interface.cpp element indicates that the service contract is
141 conversational, as described in the Assembly Specification [ASSEMBLY].

142

143 A non-conversational service, the default when no annotation is specified, indicates that the service
144 contract is stateless between requests. A conversational service indicates that requests to the service are
145 correlated.

146 2.3 Component Implementation Scopes

147 Component implementations can either manage their own state or allow the SCA runtime to do so. In the
148 latter case, SCA defines the concept of implementation scope, which specifies the visibility and lifecycle
149 contract an implementation has with the runtime. Invocations on a service offered by a component will be
150 dispatched by the SCA runtime to an implementation instance according to the semantics of its scope.

151

152 Scopes are specified using the *scope* attribute of the implementation.cpp element.

153

154 When a scope is not specified on an implementation class, the SCA runtime will interpret the
155 implementation scope as *stateless*.

156 The SCA C++ Client and Implementation Model mandates support for the four basic scopes; *stateless*,
157 *request*, *conversation*, and *composite*. Additional scopes MAY be provided by SCA runtimes.

158

159 The following snippet shows the component type for a composite scoped component:

160

161

162

163

164

```
<component name="LoanService">
  <implementation.cpp library="loan" header="LoanServiceImpl.h"
    scope="composite"/>
</component>
```

165
166 For *stateless* scoped implementations, the SCA runtime MUST prevent concurrent execution of methods
167 on an instance of that implementation. However, *composite* scoped implementations MUST be able to
168 handle multiple threads running its methods concurrently.

169
170 The following sections specify the scopes an SCA runtime MUST support for C++ component
171 implementations.

172 **2.3.1 Stateless scope**

173 For stateless implementations, a different instance may be used to service each request. Implementation
174 instances may be created or drawn from a pool of instances.

175 **2.3.2 Request scope**

176 Service requests are delegated to the same implementation instance for all colocated service invocations
177 that occur while servicing a remote service request. The lifecycle of a request scope extends from the
178 point a request on a remotable interface enters the SCA runtime and a thread processes that request until
179 the thread completes synchronously processing the request.

180
181 There are times when a local request scoped service is called without a remotable service earlier in the
182 call stack, such as when a local service is called from a non-SCA entity. In these cases, a remote request
183 is always considered to be present, but the lifetime of the request is implementation dependent. For
184 example, a timer event could be treated as a remote request.

185 **2.3.3 Composite scope**

186 All service requests are dispatched to the same implementation instance for the lifetime of the containing
187 composite. The lifetime of the containing composite is defined as the time it becomes active in the
188 runtime to the time it is deactivated, either normally or abnormally.

189
190 A composite scoped implementation may also specify eager initialization using the *eagerInit="true"*
191 attribute on the implementation.cpp element of a component definition. When marked for eager
192 initialization, the composite scoped instance will be created when its containing component is started.

193 **2.3.4 Conversation scope**

194 A conversation is defined as a series of correlated interactions between a client and a target service. A
195 conversational scope starts when the first service request is dispatched to an implementation instance
196 offering the target service. A conversational scope completes after an end operation defined by the
197 service contract is called and completes processing or the conversation expires (see [Methods that End
198 the Conversation](#)). A conversation is potentially long-running and the SCA runtime MAY choose to
199 passivate implementation instances. If this occurs, the runtime MUST guarantee implementation instance
200 state is preserved.

201
202 A conversational scoped class MUST NOT expose a service using a non-conversational interface.

203
204 Note that in the case where a conversational service is implemented by a C++ implementation that is
205 marked as conversation scoped, the SCA runtime will transparently handle implementation state. It is also
206 possible for an implementation to manage its own state. For example, a C++ implementation class having
207 a stateless (or other) scope could implement a conversational service.

208 2.4 Implementing a Configuration Property

209 Component implementations can be configured through properties. The properties and their types (not
210 their values) are defined in the component type file. The C++ component can retrieve the properties using
211 the `getProperties()` on the `ComponentContext` class.

212

213 The following code extract shows how to get the property values.

214

```
215 #include "ComponentContext.h"  
216 using namespace osoa::sca;  
217  
218 void clientMethod()  
219 {  
220     ...  
221  
222     ComponentContext context = ComponentContext::getCurrent();  
223  
224     DataObjectPtr properties = context.getProperties();  
225  
226     long loanRating = properties->getInteger("maxLoanValue");  
227  
228     ...  
229 }
```

230 2.5 Component Type and Component

231 For a C++ component implementation, a component type is specified in a side file. By default, the
232 `componentType` side file is in the same composite directory as the header file for the implementation
233 class with a name matching this header file. The location and name can be modified as described below.

234

235 This Client and Implementation Model for C++ extends the SCA Assembly model **[ASSEMBLY]** providing
236 support for the C++ interface type system and support for the C++ implementation type.

237

238 The following snippets show the C++ service interface and the C++ implementation class of a C++
239 service.

240

```
241 // LoanService interface  
242 class LoanService {  
243 public:  
244     virtual bool approveLoan(unsigned long customerNumber,  
245 unsigned long loanAmount) = 0;  
246 };
```

247

248 Implementation declaration header file.

249

```
250 class LoanServiceImpl : public LoanService  
251 {  
252 public:  
253     LoanServiceImpl();  
254     virtual ~LoanServiceImpl();  
255  
256     virtual bool approveLoan(unsigned long customerNumber,  
257 unsigned long loanAmount);  
258 };
```

259

260 Implementation.

261

```
262 #include "LoanServiceImpl.h"
263
264 ///////////////////////////////////////////////////////////////////
265 // Construction/Destruction
266 ///////////////////////////////////////////////////////////////////
267 LoanServiceImpl::LoanServiceImpl()
268 {
269     ...
270 }
271 LoanServiceImpl::~LoanServiceImpl()
272 {
273     ...
274 }
275 ///////////////////////////////////////////////////////////////////
276 // Implementation
277 ///////////////////////////////////////////////////////////////////
278 bool LoanServiceImpl::approveLoan(unsigned long customerNumber,
279     unsigned long loanAmount)
280 {
281     ...
282 }
```

283

284 The following snippet shows the component type for this component implementation.

285

```
286 <?xml version="1.0" encoding="ASCII"?>
287 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
288     <service name="LoanService">
289         <interface.cpp header="LoanService.h"/>
290     </service>
291 </componentType>
```

292

293 The following snippet shows the component using the implementation.

294

```
295 <?xml version="1.0" encoding="ASCII"?>
296 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
297     name="LoanComposite" >
298     ...
299
300     <component name="LoanService">
301         <implementation.cpp library="loan" header="LoanServiceImpl.h"/>
302     </component>
303 </composite>
```

304 2.5.1 Interface.cpp

305 The following snippet shows the schema for the C++ interface element used to type services and
306 references of component types.

307

```
308 <?xml version="1.0" encoding="ASCII"?>
309 <!-- interface.cpp schema snippet -->
310 <interface.cpp xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
311     header="NCName" class="Name"? remotable="boolean"?
312     callbackHeader="NCName" callbackClass="Name"? >
313
314     <method ... />*
315
```

316 `</interface.cpp>`

317

318 The **interface.cpp** element has the following **attributes**:

- 319 • **header : NCName (1..1)** – full name of the header file, including relative path from the composite
320 root. This header file describes the interface
- 321 • **class : Name (0..1)** – name of the class declaration in the header file, including any namespace
322 definition. If the header only contains one class then this class does not need to be specified.
- 323 • **callbackHeader : NCName (0..1)** – full name of the header file that describes the callback interface,
324 including relative path from the composite root.
- 325 • **callbackClass : Name (0..1)** – name of the class declaration for the call back in the callback header
326 file, including any namespace definition. If the header only contains one class then this class does not
327 need to be specified
- 328 • **remotable : boolean (0..1)** – indicates whether the service is remotable or local. The default is local.

329

330 The **interface.cpp** element has the following **child element**:

331 **method : CPPMethod (0..n)** – see Method

332 2.5.2 Method

333 Some methods of an interface have behavioral characteristics, which will be described later, that need to
334 be identified. This is done using a method child element of interface.cpp

335

336 The following snippet shows the interface.cpp schema with the schema for a method child element:

337

```
338 <?xml version="1.0" encoding="ASCII"?>  
339 <!-- Method schema snippet -->  
340 <interface.cpp xmlns=http://docs.oasis-open.org/ns/opencsa/sca/200712 ... >  
341  
342     <method name="NCName" oneWay="Boolean"? endsConversation="Boolean"? /*>  
343  
344 </interface.cpp>
```

345

346 The **method** element has the following **attributes**:

- 347 • **name : NCName (1..1)** – name of the method being decorated.
- 348 • **oneWay : boolean (0..1)** – see Non-blocking Calls
- 349 • **endsConversation : boolean (0..1)** – see Methods that End the Conversation

350 2.5.3 Implementation.cpp

351 The following snippet shows the schema for the C++ implementation element used to define the
352 implementation of a component..

353

```
354 <?xml version="1.0" encoding="ASCII"?>  
355 <!-- implementation.cpp schema snippet -->  
356 <implementation.cpp xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
357     library="NCName" path="Name"? header="NCName" class="Name"?  
358     scope="scope"? componentType="NCName"? allowsPassByReference=Boolean"?  
359     eagerInit="boolean"? conversationMaxAge="string"?  
360     conversationMaxIdle="string"? conversationSinglePrincipal="Boolean"? >  
361  
362     <method ... /*>  
363
```

364 `</implementation.cpp>`

365

366 The **implementation.cpp** element has the following **attributes**:

- 367 • **library : NCName (1..1)** – name of the dll or shared library that holds the factory for the service
368 component. This is the root name of the library. On Windows the suffix “.dll” will be appended to this
369 name. On linux the prefix “lib” and the suffix “.so” will be added.
- 370 • **path : Name (0..1)** - path to the library which is either absolute or relative to the root of the composite.
- 371 • **header : NCName (1..1)** – name of the header file that declares the implementation class of the
372 component. A path is optional which is relative to the root of the composite.
- 373 • **class : Name (0..1)** – name of the class declaration of the implementation, including any namespace
374 definition. If the header only contains one class then this class does not need to be specified, and the
375 header file name SHOULD be the same as the class name.
- 376 • **scope : CPPImplementationScope (0..1)** – identifies the scope of the component implementation.
377 The default is stateless.
- 378 • **componentType : NCName (0..1)** – path to the componentType file which is relative to the root of
379 the composite.
- 380 • **allowsPassByReference : boolean (0..1)** – indicates the service allows pass by reference data
381 exchange semantics on calls to it.
- 382 • **eagerInit type : boolean (0..1)** – indicates a composite scoped implementation should be initialized
383 when it is loaded.
- 384 • **conversationMaxAge : string (0..1)** – see Conversational Service Provider.
- 385 • **conversationMaxIdle : string (0..1)** – see Conversational Service Provider.
- 386 • **conversationSinglePrincipal : boolean (0..1)** – see Conversational Service Provider.

387

388 The **interface.cpp** element has the following **child element**:

389 **method : CPPImplementationMethod (0..n)** – see Implementation Method

390

391 If the class attribute is specified on implementation.cpp, then the SCA runtime MUST use class name as
392 the name of the componentType file. The SCA runtime will append “.componentType” to the class name
393 to find the componentType file.

394 2.5.4 Implementation Method

395 Some methods of an implementation have operational characteristics that need to be identified. This is
396 done using a method child element of implementation.cpp

397

398 The following snippet shows the implementation.cpp schema with the schema for a method child element:

399

```
400 <?xml version="1.0" encoding="ASCII"?>  
401 <!-- ImplementationMethod schema snippet -->  
402 <implementation.cpp xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >  
403  
404     <method name="NCName" allowsPassByReference="boolean"? /> *  
405  
406 </implementation.cpp>
```

407

408 The **method** element has the following **attributes**:

- 409 • **name : NCName (1..1)** – name of the method being decorated.

410 • ***allowsPassByReference : boolean" (0..1)*** – indicates the method allows pass by reference data
411 exchange semantics.

412 **2.6 Instantiation**

413 A C++ implementation class **MUST** be default constructable by the SCA runtime to instantiate the
414 component.

415 3 Basic Client Model

416 This section describes how to get access to SCA services from both SCA components and from non-SCA
417 components. It also describes how to call methods of these services.

418 3.1 Accessing Services from Component Implementations

419 A component can get access to a service using a component context.

420 The following snippet shows the ComponentContext C++ class with its *getService()* method.

```
421 namespace osoa {  
422     namespace sca {  
423         class ComponentContext {  
424             public:  
425                 static ComponentContextPtr getCurrent();  
426                 virtual void * getService(const std::string& referenceName)  
427                     const = 0;  
428                 ...  
429             }  
430         }  
431     }  
432 }  
433 }
```

434 The *getService()* method takes as its input argument the name of the reference and returns a pointer to
435 an object providing access to the service. The returned object will implement the abstract base class
436 definition that is used to describe the reference.
437

438

439

440 The following shows a sample of how the ComponentContext is used in a C++ component
441 implementation. The *getService()* method is called on the ComponentContext passing the reference
442 name as input. The return of the *getService()* method is cast to the abstract base class defined for the
443 reference.

444

```
445 #include "ComponentContext.h"  
446 #include "CustomerService.h"  
447  
448 using namespace osoa::sca;  
449  
450 void clientMethod()  
451 {  
452     unsigned long customerNumber = 1234;  
453     ComponentContextPtr context = ComponentContext::getCurrent();  
454     CustomerService* service =  
455     (CustomerService* )context->getService("customerService");  
456     short rating = service->getCreditRating(customerNumber);  
457 }  
458  
459  
460  
461  
462 }
```

463 **3.2 Accessing Services from non-SCA component implementations**

464 Non-SCA components can access component services by obtaining a `ComponentContextPtr` from the
465 SCA runtime and then following the same steps as a component implementation as described above.

466
467 How an SCA runtime implementation allows access to and returns a `ComponentContextPtr` is not defined
468 by this specification.

469 **3.3 Calling Service Methods**

470 The previous sections show the various options for getting access to a service. Once you have access to
471 the service, calling a method of the service is like calling a method of a C++ class.

472
473 If you have access to a service whose interface is marked as `remotable`, then on calls to methods of that
474 service you will experience remote semantics. Arguments and return are passed *by-value* and it is
475 possible to get a `ServiceUnavailableException`, which is a `ServiceRuntimeException`.

476 4 Error Handling

477 Clients calling service methods will experience business exceptions, and SCA runtime exceptions.

478

479 Business exceptions are raised by the implementation of the called service method. They should be
480 caught by client invoking the method on the service.

481

482 SCA runtime exceptions are raised by the SCA runtime and signal problems in the management of the
483 execution of components, and in the interaction with remote services. Currently the following SCA runtime
484 exceptions are defined:

- 485 • *SCAException* – defines a root exception type from which all SCA defined exceptions derive.
 - 486 – *SCANullPointerException* – signals that code attempted to dereference a null pointer from a
487 *RefCountingPointer* object.
 - 488 – *ServiceRuntimeException* - signals problems in the management of the execution of SCA
489 components.
 - 490 – *ServiceUnavailableException* – signals problems in the interaction with remote services. This
491 extends *ServiceRuntimeException*. These are exceptions that may be transient, so retrying
492 is appropriate. Any exception that is a *ServiceRuntimeException* that is not a
493 *ServiceUnavailableException* is unlikely to be resolved by retrying the operation, since it most
494 likely requires human intervention.
 - 495 – *MultipleServicesException* – signals that a method expecting identification of a single service
496 is called where there are multiple services defined. Thrown by
497 *ComponentContext::getService()*, *ComponentContext::getSelfReference()* and
498 *ComponentContext::getServiceReference()*.
 - 499 – *ConversationEndedException* – signals that a method has been called on a conversational
500 service after the conversation was ended.
 - 501 – *NoRegisteredCallbackException* – signals that a callback was invoked on a service, but a
502 callback method was not registered.

503 5 Conversational Services

504 A frequent pattern that occurs during the execution of remotable services is that a conversation is started
505 between the client of the service and the provider of the service. The conversation is a series of method
506 invocations that all pertain to a single common topic. For example, a conversation might be the series of
507 service calls that are necessary in order to apply for a bank loan.

508 5.1 Conversational Client

509 There is no special coding required by the client of a conversational service. The developer of the client
510 knows that the service is conversational from the service interface definition. The following shows an
511 example client of the conversational service described above:

512

```
513 #include "LoanApplicationClientImpl.h"  
514 #include "ComponentContext.h"  
515 #include "LoanService.h"  
516 #include "LoanApplication.h"  
517  
518 using namespace osea::sca;  
519  
520 void LoanApplicationClientImpl::clientMethod( LoanApplication loanApp,  
521 unsigned int term )  
522 {  
523     unsigned long customerNumber = 1234;  
524  
525     ComponentContextPtr context = ComponentContext::getCurrent();  
526  
527     // service is defined as member field: LoanService* service;  
528     service = (LoanService* )context->getService("loanService");  
529  
530     service->apply( loanApp );  
531     service->lockCurrentRate( term );  
532  
533 }  
534  
535 bool LoanApplicationClientImpl::isApproved()  
536 {  
537     return (service->getLoanStatus() == 1);  
538 }
```

539 5.2 Conversational Service Provider

540 A class which provides a service with a conversational interface can have any scope. In particular, it is
541 not necessary for the class to have conversation scope. However, the provider of the conversational
542 service is not required to write special code to maintain state if the implementation has conversation
543 scope since the runtime maintains state associated with the conversation, by routing each operation
544 invocation associated with the conversation to the same logical instance of the class, with state data held
545 in instance variables within the class. However, for classes with other scopes, when an operation of a
546 conversational interface is executing, the `ServiceReference::getConversationID()` returns the
547 conversation ID of the conversation. The conversation ID can be used by the class as an index to store
548 and to look up state data associated with the conversation, using some suitable storage mechanism.

549

```
550 ComponentContextPtr context = ComponentContext::getCurrent();  
551 ServiceReferencePtr serviceRef = context->getSelfReference();  
552 std::string conversationID = serviceRef->getConversationID();
```

553

554 The service implementation might also have optional conversation attributes that control the lifetime and
555 operation of the conversations it supports. These attributes are specified on the implementation.cpp
556 element:

- 557 • *maxIdleTime* - The maximum time that can pass between operations within a single conversation. If
558 more time than this passes, then the SCA runtime MAY end the conversation.
- 559 • *maxAge* - The maximum time that the entire conversation can remain active. If more time than this
560 passes, then the SCA runtime MAY end the conversation.
- 561 • *singlePrincipal* – If true, only the principal (the user) that started the conversation has authority to
562 continue the conversation.

563

564 Alternatively the conversation attributes can be specified on the implementation definition using the
565 *conversationMaxIdleTime*, *conversationMaxAge* and *conversationSinglePrincipal* of implementation.cpp.
566

567 The two attributes that take a time express the time as a string that starts with an integer, is followed by a
568 space and then one of the following: "seconds", "minutes", "hours", "days" or "years".

569

570 Not specifying timeouts means that timeout values are defined by the implementation of the SCA runtime,
571 however it chooses to do so.

572

573 Here is an example definition of a conversational service.

574

```
575 <component name="LoanService">  
576   <implementation.cpp library="loan" header="LoanServiceImpl.h"  
577     conversationMaxAge="30 days" />  
578 </component>
```

579 **5.3 Methods that End the Conversation**

580 A method of a conversational service interface can be marked with an *endsConversation="true"* attribute.
581 This means that once this method has been called, no further methods are to be called, allowing both the
582 client and the target to free up resources that were associated with the conversation. It is also possible to
583 mark a method on a callback interface (described later) as *endsConversation="true"*, in order for the
584 service provider to be the party that chooses to end the conversation. If a method is called after the
585 conversation completes a *ConversationEndedException* (which extends *ServiceRuntimeException*) is
586 thrown. This can also occur if there is a race condition between the client and the service provider calling
587 their respective *endsConversation* methods. Calling the *endConversation()* method on a service
588 reference also ends a conversation.

589

590 The following is an example implementation with a method that has been annotated as ending a
591 conversation:

592

```
593 <?xml version="1.0" encoding="ASCII"?>  
594 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">  
595   <service name="LoanService">  
596     <interface.cpp header="LoanService.h" remotable="true"  
597       requires="conversational">  
598       <method name="cancelApplication" endsConversation="true" />  
599     </interface.cpp>  
600   </service>  
601 </componentType>
```

602

603 The *cancelApplication()* operation is annotated to end the conversation.

604 5.4 Passing Conversational Services as Parameters

605 The service reference which represents a single conversation can be passed as a parameter to another
606 service, even if that other service is remote. This may be used in order to allow one component to
607 continue a conversation that had been started by another.

608 A service provider may also create a service reference for itself that it can pass to other services. A
609 service implementation does this with a call to

```
610  
611 ComponentContext::getSelfReference()
```

612
613 or

```
614  
615 ComponentContext::getSelfReference(const std::string& serviceName)
```

616
617 The second variant, which takes a *serviceName* parameter, is used if the component implements multiple
618 services.

619
620 This can be used to support complex callback patterns, such as when a callback is applicable only to a
621 subset of a larger conversation. Simple callback patterns are handled by the built-in callback support
622 described later.

623 5.5 Conversation Lifetime Summary

624 *Starting conversations*

625 Conversations start on the client side when one of the following occur:

- 626 • A service is located using `ComponentContext::getService()` or `ComponentContext::getServices()`.
- 627 • A service reference is obtained using `ComponentContext::getServiceReference()` or
628 `ComponentContext::getServiceReferences()`.

629 630 *Continuing conversations*

631 The client can continue an existing conversation, by:

- 632 • Holding the service reference that was created when the conversation started.
- 633 • Getting the service reference object passed as a parameter from another service, even remotely.

634 635 *Ending conversations*

636 A conversation ends, and any state associated with the conversation is freed up, when:

- 637 • A service operation marked `endsConversation="true"` is called.
- 638 • The service calls a method marked `endsConversation="true"` on a callback interface.
- 639 • The service's conversation lifetime timeout occurs.
- 640 • The client calls `ServiceReference::endConversation()`.
- 641 • The client calls `ServiceReference::setConversationID()` which implicitly ends any active conversation.

642
643 If a method is invoked on a service reference after a method marked `endsConversation="true"` has been
644 called then a new conversation will automatically be started. If

645 `ServiceReference::getConversationID()` is called after a method marked
646 `endsConversation="true"` is called, but before the next conversation has been started, it will return null.

647

648 If a service reference is used after the service provider's conversation timeout has caused the
649 conversation to be ended, then `ConversationEndedException` will be thrown. In order to use that service
650 reference for a new conversation, its `endConversation()` method has to be called.

651 **5.6 Application Specified Conversation IDs**

652 It is also possible to take advantage of the state management aspects of conversational services while
653 using a client-provided conversation ID. To do this, the client would use the
654 `ServiceReference::setConversationID()` method.

655

```
656 ComponentContextPtr ctx = ComponentContext::getCurrent();  
657  
658 std::string conversationID("myID");  
659  
660 ServiceReferencePtr serviceReference =  
661     ctx->getServiceReference("loanService");  
662     serviceReference->setConversationID(conversationID);  
663  
664 LoanService* service = (LoanService*)serviceReference->getService();
```

665

666 The ID **MUST** be unique to the client component over all time. If the client is not an SCA component, then
667 the ID **MUST** be globally unique.

668

669 Not all conversational service bindings support application-specified conversation IDs.

670 **5.7 Accessing Conversation IDs from Clients**

671 Whether the conversation ID is chosen by the client or is generated by the system, the client accesses
672 the conversation ID of a conversation by calling the `ServiceReference::getConversationID()`
673 method on the service reference for the conversation.

674

675 If the conversation ID is not application specified, then the `getConversationID()` method is only
676 guaranteed to return a valid value after the first operation has been invoked, otherwise it returns an empty
677 string.

688 6 Asynchronous Programming

689 Asynchronous programming of a service is where a client invokes a service and carries on executing
690 without waiting for the service to execute. Typically, the invoked service executes at some later time.
681 Output from the invoked service, if any, is fed back to the client through a separate mechanism, since no
682 output is available at the point where the service is invoked. This is in contrast to the call-and-return style
683 of synchronous programming, where the invoked service executes and returns any output to the client
684 before the client continues. The SCA asynchronous programming model consists of support for non-
685 blocking method calls, callbacks, and conversational services. Each of these topics is discussed in the
686 following sections.

687 6.1 Non-blocking Calls

688 Non-blocking calls represent the simplest form of asynchronous programming, where the client of the
689 service invokes the service and continues processing immediately, without waiting for the service to
690 execute.

691

692 Any method that returns "void" and has no declared exceptions can be marked by using the
693 *oneWay="true"* attribute in the interface definition of the service. This means that the method is non-
694 blocking and the SCA runtime MAY use a binding that buffers the requests and sends it at some later
695 time.

696

697 The following snippet shows the component type for a service with the *reportEvent()* method declared as
698 a one-way method:

699

```
700 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">  
701   <service name="LoanService">  
702     <interface.cpp header="LoanService.h">  
703       <method name="reportEvent" oneWay="true" />  
704     </interface.cpp>  
705   </service>  
706 </componentType>
```

707

708 SCA does not currently define a mechanism for making non-blocking calls to methods that return values
709 or are declared to throw exceptions. It is recommended that service designers define one-way methods
710 as often as possible, in order to give the greatest degree of binding flexibility to deployers.

711 6.2 Callbacks

712 Callback services are used by *bidirectional services* as defined in the Assembly Specification
713 **[ASSEMBLY]**.

714

715 A callback interface is declared by the *callbackHeader* and *callbackClass* attributes in the interface
716 definition of the service. The following snippet shows the component type for a service *MyService* with the
717 interface defined in *MyService.h* and the interface for callbacks defined in *MyServiceCallback.h*,

718

```
719 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" >  
720   <service name="MyService">  
721     <interface.cpp header="MyService.h"  
722       callbackHeader="MyServiceCallback.h"/>  
723   </service>  
724 </componentType>
```


725 6.2.1 Stateful Callbacks

726 A stateful callback represents a specific implementation instance of the component that is the client of the
727 service. The interface of a stateful callback normally is *conversational*.

728 A component gets access to the callback service by using the `getCallback()` method of the
729 `ServiceReference` (obtained from the `ComponentContext`).

730

731 The following is an example service implementation for the service and callback declared above. When
732 the `someMethod` has completed its processing it retrieves the callback service from the `ServiceReference`
733 and invokes a callback method.

734

```
735 #include "MyServiceImpl.h"  
736 #include "MyServiceCallback.h"  
737 #include "osoa/sca/ComponentContext.h"  
738 using namespace osoa::sca;  
739  
740 MyServiceImpl::someMethod( unsigned int arg )  
741 {  
742     ...  
743     // do some processing...  
744  
745     ComponentContextPtr context = ComponentContext::getCurrent();  
746     ServiceReferencePtr serviceRef = context->getSelfReference();  
747     MyServiceCallback* callback = (MyServiceCallback* ) serviceRef-  
748     >getCallback();  
749     callback->receiveResult(result);  
750 }
```

751

752 The following shows how a client component would to invoke the `MyService` service and receive the
753 callback.

754

```
755 #include "MyServiceImpl.h"  
756 #include "MyServiceCallback.h"  
757 #include "osoa/sca/ComponentContext.h"  
758 using namespace osoa::sca;  
759  
760 void clientMethod( unsigned int arg )  
761 {  
762     // locate the service  
763     ComponentContextPtr context = ComponentContext::getCurrent();  
764     MyService* service = (MyService*) context->getService("myservice");  
765     service->someMethod(arg);  
766 }  
767  
768 MyServiceCallback::receiveResult(unsigned int result)  
769 {  
770     // code to process result  
771 }
```

772

773 Stateful callbacks support some of the same use cases as are supported by the ability to pass service
774 references as parameters. The primary difference is that stateful callbacks do not require that any
775 additional parameters be passed with service operations. This can be a great convenience. If the service
776 has many operations and any of those operations could be the first operation of the conversation, it would
777 be unwieldy to have to take a callback parameter as part of every operation, just in case it is the first
778 operation of the conversation. It is also more natural than requiring the application developers to invoke
779 an explicit operation whose only purpose is to pass the callback object to be used.

780 6.2.2 Stateless Callbacks

781 A stateless callback interface is a callback whose interface is not *conversational*. Unlike stateful services,
782 the client that uses stateless callbacks will not have callback methods routed to an instance of the client
783 that contains any state that is relevant to the conversation. As such, it is the responsibility of such a client
784 to perform any persistent state management itself. The only information that the client has to work with
785 (other than the parameters of the callback method) is a callback ID that is passed with requests to the
786 service and is guaranteed to be returned with any callback. The callback ID is retrieved from the
787 `ServiceReference`.

788

789 The following snippets show a client setting a callback id before invoking the asynchronous service and
790 the callback method retrieving the callback ID:

791

```
792 void clientMethod( unsigned int arg )  
793 {  
794     // locate the service  
795     ComponentContextPtr context = ComponentContext::getCurrent();  
796     ServiceReferencePtr svcRef = context->getServiceReference("myservice");  
797     svcRef->setCallbackID("1234");  
798     MyService* service = (MyService*)svcRef->getService();  
799     service->someMethod(arg);  
800 }  
801  
802  
803 MyServiceCallback::receiveResult(unsigned int result)  
804 {  
805     ComponentContextPtr context = ComponentContext::getCurrent();  
806  
807     ServiceReferencePtr serviceRef = context->getSelfReference();  
808     std::string id = serviceRef->getCallbackID();  
809  
810     // code to process result  
811 }
```

812 6.2.3 Implementing Multiple Bidirectional Interfaces

813 Since it is possible for a single class to implement multiple services, it is also possible for callbacks to be
814 defined for each of the services that it implements. To access the callbacks the
815 `ServiceReference::getCallback(serviceName)` method must be used, passing in the name of the service
816 for which the callback is to be obtained.

817 6.2.4 Customizing the Callback Identity

818 The identity that is used to identify a callback request is, by default, generated by the SCA runtime.
819 However, it is possible to provide an application specified identity to be used to identify the callback by
820 calling the `ServiceReference::setCallbackID()` method. This can be used in either stateful or
821 stateless callbacks. The identity will be sent to the service provider, and the binding **MUST** guarantee
822 that the SCA runtime will send the ID back when any callback method is invoked.

823

824 The callback ID has the same restrictions as the conversation ID. It **MUST** be unique to the client
825 component over all time or it **MUST** be globally unique if the client is nto an SCA component. Bindings
826 determine the particular mechanisms to use for transmission of the identity and these may lead to further
827 restrictions when using a given binding.

828 7 C++ API

829 All the C++ interfaces are found in the namespace `osa::sca`, which has been omitted from the following
830 descriptions for clarity.

831 7.1 Reference Counting Pointers

832 These are a derived version of the familiar smart-pointer. The pointer class holds a real (dumb) pointer to
833 the object. If the reference counting pointer is copied, then a duplicate pointer is returned with the same
834 real pointer. A reference count within the object is incremented for each copy of the pointer, so only when
835 all pointers go out of scope will the object be freed.

836 `RefCountingPointer` defines methods raw pointer like semantics. This includes defining operators for
837 dereferencing the pointer (`*`, `->`), as well as operators for determining the validity of the pointer.

838

```
839 template <typename T>  
840 class RefCountingPointer {  
841 public:  
842     T& operator* () const;  
843     T* operator-> () const;  
844     operator void* () const;  
845     bool operator! () const;  
846 };
```

847

848 The `RefCountingPointer` class has the following methods:

- 849 • `operator*()` – Dereferences the underlying pointer, returning a reference to the value. This is
850 equivalent to calling `*p` where `p` is the underlying pointer. If this method is invoked on null pointer, an
851 `SCANullPointerException` is thrown.
- 852 • `operator->()` – Allows methods to be invoked directly on the underlying pointer. This is equivalent to
853 invoking `p->func()` where `func()` is a method defined on the underlying pointer type. If this method is
854 invoked on a null pointer, an `SCANullPointerException` is thrown.
- 855 • `operator void*()` – Returns null if the underlying pointer is null, otherwise returns a non-zero value.
856 This method allow for checking whether a `RefCountingPointer` is set, i.e. if `(p) { /* do something */ }`.
- 857 • `operator!()` – Returns true if the underlying pointer is null, false otherwise. This method allows for
858 checking wither is `RefCountingPointer` is not set, i.e. if `(!p) { /* do something */ }`

859

860 Reference counting pointers in SCA have the same name as the type they are pointing to, with a suffix of
861 `Ptr`. (E.g. `ComponentContextPtr`, `ServiceReferencePtr`).

862 7.2 Component Context

863 The following shows the `ComponentContext` interface.

864

```
865 class ComponentContext {  
866 public:  
867     static ComponentContextPtr getCurrent();  
868  
869     virtual std::string getURI() const = 0;  
870  
871     virtual void * getService(const std::string& referenceName) const = 0;  
872     virtual std::list<void*> getServices(const std::string& referenceName)  
873     const = 0;
```

```

874
875 virtual ServiceReferencePtr getServiceReference(const std::string&
876 referenceName) const = 0;
877 virtual std::list<ServiceReferencePtr> getServiceReferences(const std::string&
878 referenceName) const = 0;
879
880
881     virtual DataObjectPtr getProperties() const = 0;
882 virtual DataFactoryPtr getDataFactory() const = 0;
883
884
885 virtual ServiceReferencePtr getSelfReference() const = 0;
886 virtual ServiceReferencePtr getSelfReference(const std::string& serviceName)
887 const = 0;
888 };

```

889

890 The ComponentContext C++ interface has the following methods:

- 891 • *getCurrent()* – returns the ComponentContext for the current component
- 892 • *getURI()* – returns the absolute URI of the component.
- 893 • *getService()* – returns a pointer to object implementing the interface defined for the named reference.

894 Input to the method is the name of a reference defined on this component. A

895 MultipleServicesException will be thrown if the reference resolves to more than one service.
- 896 • *getServices()* – returns a list of objects implementing the interface defined for the named reference.

897 Input to the method is the name of a reference defined on this component.
- 898 • *getServiceReference()* – returns a ServiceReference for the specified service. A

899 MultipleServicesException will be thrown if the reference resolves to more than one service.
- 900 • *getServiceReferences()* – returns a list of ServiceReferences for the named reference.
- 901 • *getProperties()* – Returns an SDO **[SDO]** from which all the properties defined in the componentType

902 file can be retrieved.
- 903 • *getDataFactory()* – Returns an SDO DataFactory which can be used to create data objects.
- 904 • *getSelfReference()* – Returns a ServiceReference that can be used to invoke this component over the

905 designated service. The second variant, which takes a serviceName parameter, is used if the

906 component implements multiple services. A MultipleServicesException is thrown if the first variant is

907 called for a component implementing multiple services.

908 7.3 ServiceReference

909 The following shows the ServiceReference interface.

910

```

911 class ServiceReference {
912 public:
913 virtual void* getService() const = 0;
914
915     virtual std::string getConversationID() const = 0;
916 virtual void setConversationID(const std::string& id) = 0;
917
918     virtual std::string getCallbackID() const = 0;
919 virtual void setCallbackID(const std::string& id) = 0;
920
921 virtual void* getCallback() const = 0;
922
923 virtual void endConversation() = 0;
924
925 };

```

926

927 The ServiceReference interface has the following methods:

928

- 929 • *getService()* – returns a pointer to the service for this reference. A MultipleServicesException will be
- 930 thrown if the reference resolves to more than one service.
- 931 • *getConversationID()* – returns the conversation ID
- 932 • *setConversationID()* – sets a user provided conversation ID
- 933 • *getCallbackID()* – returns the callback ID
- 934 • *setCallbackID()* – sets the callback ID
- 935 • *getCallback()* – returns the callback service
- 936 • *endConversation()* – ends the conversation for this service reference

937

938 The detailed description of the usage of these methods is described in the sections on Conversational
939 Services and Asynchronous Programming in this document.

940 7.4 SCAException

941 The following shows the SCAException interface.

942

```
943 namespace osoa {  
944     namespace sca {  
945  
946         class SCAException : public std::exception {  
947         public:  
948             const char* getEClassName() const;  
949             const char* getMessageText() const;  
950             const char* getFileName() const;  
951             unsigned long getLineNumber() const;  
952             const char* getFunctionName() const;  
953         };  
954     }  
955 }
```

956

957 The SCAException C++ interface has the following methods:

- 958 • *getEClassName()* – Returns the type of the exception as a string. e.g. “ServiceUnavailableException”.
- 959 • *getMessageText()* – Returns the message which the SCA runtime attached to the exception.
- 960 • *getFileName()* – Returns the filename within which the exception occurred – Will be zero if the
961 filename is not known.
- 962 • *getLineNumber()* – Returns the line number at which the exception occurred.
- 963 • *getFunctionName()* – Returns the function name in which the exception occurred.

964

965 Details concerning this class and its derived types are described in the section on Error Handling in this
966 document.

967 7.5 SCANullPointerException

968 The following shows the SCANullPointerException interface.

969

```
970 namespace osoa {
```

```
971     namespace sca {
972
973         class SCANullPointerException : public SCAException {
974             };
975     }
976 }
```

977

978 **7.6 ServiceRuntimeException**

979 The following shows the ServiceRuntimeException interface.

980

```
981 namespace osoa {
982     namespace sca {
983
984         class ServiceRuntimeException : public SCAException {
985             };
986     }
987 }
```

988 **7.7 ServiceUnavailableException**

989 The following shows the ServiceUnavailableException interface.

990

```
991 namespace osoa {
992     namespace sca {
993
994         class ServiceUnavailableException : public ServiceRuntimeException {
995             };
996     }
997 }
```

998 **7.8 NoRegisteredCallbackException**

999 The following shows the NoRegisteredCallbackException interface.

1000

```
1001 namespace osoa {
1002     namespace sca {
1003
1004         class NoRegisteredCallbackException : public
1005 ServiceRuntimeException {
1006             };
1007     }
1008 }
```

1009

1010 **7.9 ConversationEndedException**

1011 The following shows the ConversationEndedException interface.

1012

```
1013 namespace osoa {
1014     namespace sca {
1015
1016         class ConversationEndedException : public ServiceRuntimeException {
1017             };

```

```
1018     }  
1019 }
```

1020 **7.10 MultipleServicesException**

1021 The following shows the MultipleServicesException interface.

1022

```
1023 namespace ooa {  
1024     namespace sca {  
1025         class MultipleServicesException : public ServiceRuntimeException {  
1026             };  
1027     }  
1028 }  
1029 }
```

1030

8 WSDL to C++ and C++ to WSDL Mapping

1031

The SCA Client and Implementation Model for C++ applies the *WSDL to C++* mapping rules as defined by the OMG *WSDL to C++ Mapping Specification [CPPWSDL]* and the C++ to WSDL mapping rules as

1032

defined in [C++ to WSDL Mapping](#).

1033

9 Packaging

9.1 Composite Packaging

The physical realization of an SCA composite is a folder in a file system containing at least one .composite file. The following shows the MyValueComposite just after creation in a file system.

```
MyValue/  
    MyValue.composite
```

Besides the .composite file the composite contains artifacts that define the implementations of components, and that define the bindings of services and references. Examples of artifacts would C++ header files, shared libraries (for example, dll), WSDL portType definitions, XML schemas, WSDL binding definitions, and so on. These artifacts can be contained in subfolders of the composite, whereby programmers have the freedom to construct a folder structure that best fits the needs of their project. The following shows the complete MyValue composite folder file structure in a file system.

```
MyValue/  
    MyValue.composite  
  
    bin/  
        myvalue.dll  
  
    services/myValue/  
        MyValue.h  
        MyValueImpl.h  
        MyValueImpl.componentType  
        MyValueService.wsdl  
  
    services/customer/  
        CustomerService.h  
        CustomerServiceImpl.h  
        Customer.h  
  
    services/stockquote/  
        StockQuoteService.h  
        StockQuoteService.wsdl
```

Note that the folder structure is not architected, other than the .composite file MUST be at the root of the folder structure.

Addressing of the resources inside of the composite is done relative to the root of the composite (i.e. the location of the .composite file).

Shared libraries (including dlls) will be located as specified in the <implementation.cpp> element in the .composite file relative to the root of the composite.

XML definitions like XML schema complex types or WSDL portTypes are referenced by composite and component type files using URIs. These URIs consist of the namespace and local name of these XML definitions. The composite folder or one of its subfolders has to contain the XML resources providing the XML definitions identified by these URI's.

1080 **10 Types Supported in Service Interfaces**

1081 A service interface can support a restricted set of the types available to a C++ programmer. This section
1082 summarizes the valid types that can be used.

1083 **10.1 Local service**

1084 For a local service the types that are supported are:

- 1085 • Any of the C++ primitive types (for example, int, short, char). In this case the types will be passed by
1086 value as is normal for C++.
- 1087 • Pointers to any of the C++ primitive types (for example, int *, short *, char *).
- 1088 • The const keyword can be used for any pointer to a C++ primitive type (for example const char *). If
1089 this is used on a parameter then the destination may not change the value.
- 1090 • C++ class. The class will be passed by value as is normal for C++.
- 1091 • Pointer to a C++ class. A pointer will be passed to the destination which can then modify the original
1092 contents.
- 1093 • DataObjectPtr. An SDO pointer. This will be passed by reference.
- 1094 • References to C++ classes (passed by reference)

1095 **10.2 Remotable service**

1096 For a remotable service being called by another service the data exchange semantics is by-value. In this
1097 case the types that are supported are:

- 1098 • Any of the C++ primitive types (for example, int, short, char). This will be copied.
- 1099 • C++ classes. These will be passed using the copy constructor. The copy constructor must make sure
1100 that any embedded references, pointers or objects are copied appropriately.
- 1101 • DataObjectPtr. An SDO pointer. The SDO will be copied and passed to the destination.

1102 Unless the interface is marked as allowing pass by reference semantics, the behavior of the following are
1103 not defined:

- 1104 • Pointers.
- 1105 • References.

1106 **11 Restrictions on C++ header files**

1107 A C++ header file that is used to describe an interface has some restrictions. It **MUST**:

- 1108 • Declare at least one class with:
 - 1109 – At least one public method.
 - 1110 – All public methods must be pure virtual (virtual with no implementation)

1111

1112 The following C++ keywords and constructs **MUST NOT** be used:

- 1113 • Macros
- 1114 • inline methods
- 1115 • friend classes

1116 12C++ to WSDL Mapping

1117 This section describes a mapping from C++ header interfaces to a WSDL description of that interface.
1118 The intent is for implementations of this proposal to be able to deploy a service based only on a C++
1119 header definition and for a WSDL definition of that service to be generated from the C++, either at deploy
1120 or run time.

1121
1122 This mapping currently only deals with producing document/literal wrapped style services and WSDL from
1123 C++ header files. Support for additional styles, if provided SHOULD be consistent with the mapping
1124 specified in this specification. Support for additional styles is implementation dependent.

1125 12.1 Parameter and Return Type mappings

1126 This section details how types used as parameters or return types in C++ method prototypes get mapped
1127 to XML schema elements in the generated WSDL.

1128 12.1.1 Built-in, STL and SDO type mappings

<i>C++ built in, STL and SDO types</i>	<i>Notes</i>	<i>XML Type</i>
bool		xsd:boolean
char	signed 8-bit ¹	xsd:byte
unsigned char	unsigned 8-bit ¹	xsd:unsignedByte
short	signed 16-bit ¹	xsd:short
unsigned short	unsigned 16-bit ¹	xsd:unsignedShort
int	signed 16-bit ¹	xsd:short
unsigned int	unsigned 16-bit ¹	xsd:unsignedShort
long	signed 32-bit ¹	xsd:int
unsigned long	unsigned 32-bit ¹	xsd:unsignedInt
long long	signed 64-bit ¹	xsd:long
unsigned long long	unsigned 64-bit ¹	xsd:unsignedLong
float	32-bit floating point (IEEE-754-1985) ¹	xsd:float
double	64-bit floating point (IEEE-754-1985) ¹	xsd:double
long double	64-bit floating point (platform dependent, IEEE-754-1985) ¹	xsd:double
char* or char array	A null-terminated UTF-8 encoded string	xsd:string
wchar_t* or wchar_t array ²	A null-terminated UTF-16 or UTF-32	xsd:string

¹ The size of this type is not fixed according to the C++ standard. The size indicated is the minimum size required by the C++ specification.

	encoded string	
std::string	A UTF-8 encoded string	xsd:string
std::wstring ²	A UTF-16 or UTF-32 encoded string	xsd:string
Commonj::sdo::DataObjectPtr		xsd:any

1129

1130 For example, a C++ method prototype defined in a header such as:

1131

```
1132 long myMethod(char* name, int id, double value);
```

1133

1134 would generate a schema like:

1135

```
1136 <xsd:element name="myMethod">
1137   <xsd:complexType>
1138     <xsd:sequence>
1139       <xsd:element name="name" type="xsd:string"/>
1140       <xsd:element name="id" type="xsd:short"/>
1141       <xsd:element name="value" type="xsd:double"/>
1142     </xsd:sequence>
1143   </xsd:complexType>
1144 </xsd:element>
1145
1146 <xsd:element name="myMethodResponse">
1147   <xsd:complexType>
1148     <xsd:sequence>
1149       <xsd:element name="myMethodResponseData" type="xsd:int"/>
1150     </xsd:sequence>
1151   </xsd:complexType>
1152 </xsd:element>
```

1153 12.1.2 Binary data mapping

1154 Binary data, such as data passed via non-null-terminated char* or char arrays, is not supported in this
 1155 mapping. char* and char array parameters and return types are always mapped to xsd:string, and are
 1156 null-terminated. This requirement also applies to wchar_t* and wchar_t array parameters.

1157 12.1.3 Array mapping

1158 C++ arrays passed in or out of methods get mapped as normal elements but with multiplicity allowed via
 1159 the minOccurs and maxOccurs attributes. E.g. a method prototype such as

1160

```
1161 long myMethod(char* name, int idList[], double value);
```

1162

1163 would generate a schema like:

1164

```
1165 <xsd:element name="myMethod">
1166   <xsd:complexType>
1167     <xsd:sequence>
1168       <xsd:element name="name" type="xsd:string"/>
```

² The encoding associated with a wchar_t variable is determined by the size of the wchar_t type. If wchar_t is a 16-bit type, UTF-16 is used, otherwise UTF-32 is used.

```

1169     <xsd:element name="idList" type="xsd:short"
1170                 minOccurs="0" maxOccurs="unbounded"/>
1171     <xsd:element name="value" type="xsd:double"/>
1172 </xsd:sequence>
1173 </xsd:complexType>
1174 </xsd:element>

```

1175 12.1.4 Multi-dimensional array mapping

1176 Multi-dimensional arrays will need converting into nested elements. E.g. a method prototype such as

1177

```
1178 long myMethod(int multiIdArray[][4][2]);
```

1179

1180 would generate a schema like:

1181

```

1182 <xsd:element name="myMethod">
1183   <xsd:complexType>
1184     <xsd:sequence>
1185       <xsd:element name="multiIdArray"
1186                   minOccurs="0" maxOccurs="unbounded"/>
1187       <xsd:complexType>
1188         <xsd:sequence>
1189           <xsd:element name="multiIdArray"
1190                       minOccurs="4" maxOccurs="4"/>
1191           <xsd:complexType>
1192             <xsd:sequence>
1193               <xsd:element name="multiIdArray" type="xsd:short"
1194                           minOccurs="2" maxOccurs="2" />
1195             </xsd:sequence>
1196           </xsd:complexType>
1197         </xsd:element>
1198       </xsd:sequence>
1199     </xsd:complexType>
1200   </xsd:element>
1201 </xsd:sequence>
1202 </xsd:complexType>
1203 </xsd:element>

```

1204 12.1.5 Pointer/reference mapping

1205 A C++ method prototype that uses the 'pass-by-reference' style, defining parameters that are either
1206 references or pointers, is not meaningful when applied to web services, which rely on serialized data. A
1207 C++ method prototype that uses references or pointers will be converted to a WSDL operation that is
1208 defined as if the parameters were 'pass-by-value', with the web-service implementation framework
1209 responsible for creating the value, obtaining its pointer and passing that to the implementation class.

1210 E.g. a C++ method prototype defined in a header such as:

1211

```
1212 long myMethod(char* name, int* id, double* value);
```

1213

1214 would generate a schema like:

1215

```

1216 <xsd:element name="myMethod">
1217   <xsd:complexType>
1218     <xsd:sequence>
1219       <xsd:element name="name" type="xsd:string"/>
1220       <xsd:element name="id" type="xsd:short"/>
1221       <xsd:element name="value" type="xsd:double"/>

```

```
1222     </xsd:sequence>
1223     </xsd:complexType>
1224 </xsd:element>
```

1225

1226 Note here how the char* type is a special case – char* parameters map to xsd:string.

1227 References and pointers are also used where in/out parameters are required – where the method
1228 changes the value of the parameter and those changes are subsequently available in the invoking code –
1229 see In/Out Parameters below.

1230 **12.1.6 STL container mapping**

1231 A C++ method prototype that uses STL containers (std::vector, std::list, std::map, std::set, etc) as
1232 parameters or return types can be converted to a WSDL operation if the container is defined as holding
1233 types that can be mapped. For example, a method such as:

1234

```
1235     std::string myMethod( DataMap myMap, int id );
```

1236

1237 with the DataMap type defined as an STL container holding mappable types:

1238

```
1239     typedef std::map<std::string, double> DataMap;
```

1240

1241 could convert to a schema like:

1242

```
1243     <xsd:element name="myMethod">
1244     <xsd:complexType>
1245     <xsd:sequence>
1246     <xsd:element name="myMap" type="DataMap"
1247     minOccurs="0" maxOccurs="unbounded"/>
1248     <xsd:element name="id" type="xsd:short"/>
1249     </xsd:sequence>
1250     </xsd:complexType>
1251 </xsd:element>
1252
1253     <xsd:complexType name="DataMap">
1254     <xsd:sequence>
1255     <xsd:element name="data1" type="xsd:string"/>
1256     <xsd:element name="data2" type="xsd:double"/>
1257     </xsd:sequence>
1258 </xsd:complexType>
```

1259 **12.1.7 Struct mapping**

1260 C style structs that contain types that can be mapped, are themselves mapped to complex types. For
1261 example, a method such as:

1262

```
1263     std::string myMethod( DataStruct data, int id );
```

1264

1265 with the DataStruct type defined as a struct holding mappable types:

1266

```
1267     struct DataStruct {
1268     std::string name;
1269     double value;
1270 };
```

1271

1272 could convert to a schema like:

1273

```
1274 <xsd:element name="myMethod">
1275   <xsd:complexType>
1276     <xsd:sequence>
1277       <xsd:element name="data" type="DataStruct" />
1278       <xsd:element name="id" type="xsd:short"/>
1279     </xsd:sequence>
1280   </xsd:complexType>
1281 </xsd:element>
1282
1283 <xsd:complexType name="DataStruct">
1284   <xsd:sequence>
1285     <xsd:element name="name" type="xsd:string"/>
1286     <xsd:element name="value" type="xsd:double"/>
1287   </xsd:sequence>
1288 </xsd:complexType>
```

1289

1290 Handling of C++ style structs is not defined by this specification and is implementation dependent. In
1291 particular, C++ style structs that have protected or private data, or which require construction/destruction
1292 semantics may not be supported.

1293 **12.1.8 Enum mapping**

1294 In C++ enums define a list of named symbols that map to values. If a method uses an enum type, this can
1295 be mapped to a restricted element in the WSDL schema.

1296 For example, a method such as:

1297

```
1298 std::string getValueFromType( ParameterType type );
```

1299

1300 with the ParameterType type defined as an enum:

1301

```
1302 enum ParameterType
1303 {
1304     UNSET = 1,
1305     TYPEA,
1306     TYPEB,
1307     TYPEC
1308 };
```

1309

1310 could convert to a schema like:

1311

```
1312 <xsd:element name="getValueFromType">
1313   <xsd:complexType>
1314     <xsd:sequence>
1315       <xsd:element name="type" type="ParameterType"/>
1316     </xsd:sequence>
1317   </xsd:complexType>
1318 </xsd:element>
1319
1320 <xsd:simpleType name="ParameterType">
1321   <xsd:restriction base="xsd:int">
1322     <xs:minInclusive value="1"/>
1323     <xs:maxInclusive value="4"/>
1324   </xsd:restriction>
```


1325 `</xsd:simpleType>`

1326

1327 The restriction used will have to be appropriate to the values of the enum elements. For example, a non-
1328 contiguous enum like:

1329

```
1330 enum ParameterType  
1331 {  
1332     UNSET = 'u',  
1333     TYPEA = 'A',  
1334     TYPEB = 'B',  
1335     TYPEC = 'C'  
1336 };
```

1337

1338 could convert to a schema like:

1339

```
1340 <xsd:simpleType name="ParameterType">  
1341   <xsd:restriction base="xsd:int">  
1342     <xsd:enumeration value="86"/> <!-- Character 'u' -->  
1343     <xsd:enumeration value="65"/> <!-- Character 'A' -->  
1344     <xsd:enumeration value="66"/> <!-- Character 'B' -->  
1345     <xsd:enumeration value="67"/> <!-- Character 'C' -->  
1346   </xsd:restriction>  
1347 </xsd:simpleType>
```

1348 **12.1.9 Union mapping**

1349 In C++ unions allow the same memory location to be used for different variables. Handling of C++ unions
1350 is not defined by this mapping, and is implementation dependent. For portability it is recommended that
1351 unions not be used in service interfaces.

1352 **12.1.10 Typedef mapping**

1353 Typedef mappings are supported by this specification, and will be followed when evaluating parameter
1354 and return types. This mapping does not define whether typedef names will be used in the resulting
1355 WSDL file. The use of these names is implementation dependent.

1356 **12.1.11 Pre-processor mapping**

1357 C++ allows for the use of pre-processor directives in order to control how a C++ header is parsed.
1358 Handling for pre-processor directives is not defined by this mapping, and support is implementation
1359 dependent. For portability it is recommended that pre-processor directives not be used in service
1360 interfaces.

1361 **12.1.12 Nesting types**

1362 If a struct, enum or STL container nests other structs, enums or STL containers within itself, it is mapped,
1363 as long as the nesting eventually boils down to a mappable type. For example, a method such as:

1364

```
1365     std::string myMethod(DataStruct data);
```

1366

1367 with types defined as follows:

1368

```
1369 struct DataStruct {  
1370     std::string name;  
1371     ValuesVector values;
```

```

1372     ParameterType type;
1373 };
1374
1375 typedef std::vector<double> ValuesVector;
1376
1377 enum ParameterType
1378 {
1379     UNSET = 1,
1380     TYPEA,
1381     TYPEB,
1382     TYPEC
1383 };

```

1384

1385 would convert to a schema like:

1386

```

1387 <xsd:element name="myMethod">
1388   <xsd:complexType>
1389     <xsd:sequence>
1390       <xsd:element name="data" type="DataStruct"/>
1391     </xsd:sequence>
1392   </xsd:complexType>
1393 </xsd:element>
1394
1395 <xsd:complexType name="DataStruct">
1396   <xsd:sequence>
1397     <xsd:element name="name" type="xsd:string"/>
1398     <xsd:element name="values" type="ValuesVector"/>
1399     <xsd:element name="type" type="ParameterType"/>
1400   </xsd:sequence>
1401 </xsd:complexType>
1402
1403 <xsd:complexType name="ValuesVector">
1404   <xsd:sequence>
1405     <xsd:element name="data" type="xsd:double"/>
1406   </xsd:sequence>
1407 </xsd:complexType>
1408
1409 <xsd:simpleType name="ParameterType">
1410   <xsd:restriction base="xsd:int">
1411     <xs:minInclusive value="1"/>
1412     <xs:maxInclusive value="4"/>
1413   </xsd:restriction>
1414 </xsd:simpleType>

```

1415 **12.1.13 SDO mapping**

1416 C++ method prototypes that use `commonj::sdo::DataObjectPtr` objects as parameter or return types are
 1417 mapped to the any type in the WSDL schema as the schema for a Data Object is unknown before
 1418 runtime. For example, a C++ method prototype defined in a header such as:

1419

```

1420 long myMethod(commonj::sdo::DataObjectPtr data);

```

1421

1422 would generate a schema like:

1423

```

1424 <xsd:element name="myMethod">
1425   <xsd:complexType>
1426     <xsd:sequence>
1427       <xsd:element name="data">
1428         <xsd:complexType>

```

```
1429     <xsd:sequence>
1430         <xsd:any processContents="skip"/>
1431     </xsd:sequence>
1432 </xsd:complexType>
1433 </xsd:element>
1434 </xsd:sequence>
1435 </xsd:complexType>
1436 </xsd:element>
```

1437

1438 Typed (static) Data Objects are supported via the rules for User-defined types mapping below.

1439 **12.1.14 void* mapping**

1440 The void* type is not supported due to its undefined nature.

1441 **12.1.15 User-defined types (UDT) mapping**

1442 C++ method prototypes that employ user-defined C++ types as return types or parameters are mapped if
1443 the C++ object defines setter and getter methods for its member variables. The types of the member
1444 variables must be mappable to a schema element via the rules in this document. The names of the
1445 schema elements are defined by the set[Name] and get[Name] methods. For example, a C++ method
1446 prototype defined in a header such as:

1447

```
1448     long myMethod(AnObject data);
```

1449

1450 where AnObject is defined in a locatable C++ header as:

1451

```
1452     class AnObject
1453     {
1454     public:
1455         AnObject();
1456
1457         std::string getMyString() const;
1458         double getMyDouble() const;
1459
1460         void setMyString(std::string data);
1461         void setMyDouble(double otherData);
1462     };
```

1463

1464 would generate a schema like:

1465

```
1466     <xsd:element name="myMethod">
1467         <xsd:complexType>
1468             <xsd:sequence>
1469                 <xsd:element name="data" type="AnObject"/>
1470             </xsd:sequence>
1471         </xsd:complexType>
1472     </xsd:element>
1473
1474     <xsd:complexType name="AnObject">
1475         <xsd:sequence>
1476             <xsd:element name="MyString" type="xsd:string"/>
1477             <xsd:element name="MyDouble" type="xsd:double"/>
1478         </xsd:sequence>
1479     </xsd:complexType>
```

1480

1481 Both set[Name] and get[Name] must be present in order for the variable to be mapped for the UDT type.
1482 In addition, any UDT must provide a default constructor.

1483

1484 This specification does not define support for arrays within UDTs. Instead it is recommended that classes
1485 use STL containers to represent collections.

1486 **12.1.16 Included or Inherited types**

1487 An implementation **MUST** completely map a C++ interface to WSDL, including types (structs, enums,
1488 classes, etc) that might be defined in a different C++ header than the one that is being mapped.
1489 Implementations **SHOULD** allow a list of “include” directories to be specified. Types that are included (via
1490 a #include “SomeHeader.h” statement) or inherited from a superclass **MUST** be mappable to a schema
1491 element via the rules in this document.

1492 **12.2 Namespace mapping**

1493 Where a C++ header defines a namespaced class, the namespace and class name should map to a
1494 target namespace used in the generated WSDL. For example, a header file such as:

1495

```
1496 namespace myCorp  
1497 {  
1498     namespace myServices  
1499     {  
1500         class ExampleService  
1501         {  
1502             public:  
1503                 // Methods go here  
1504             };  
1505         }  
1506     }
```

1507

1508 would generate WSDL like:

1509

```
1510 <definitions name="ExampleService"  
1511     xmlns="http://schemas.xmlsoap.org/wsdl/"  
1512     targetNamespace="http://myCorp/myServices/ExampleService"  
1513     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"  
1514     xmlns:xsd="http://www.w3.org/2001/XMLSchema">  
1515     <types>  
1516         <xsd:schema  
1517             targetNamespace="http://myCorp/myServices/ExampleService"  
1518             xmlns:xsd="http://www.w3.org/2001/XMLSchema">  
1519         ...
```

1520

1521 Implementations **SHOULD** allow namespace mappings to be specified separately to override this default
1522 behavior.

1523 **12.3 Class mapping**

1524 A single class in a C++ header maps to a single WSDL service element, a single WSDL binding element
1525 and a single WSDL portType element. The WSDL service element contains a single WSDL port element.
1526 The WSDL binding and WSDL portType elements each contain multiple WSDL operation elements that
1527 map to the public methods defined in the C++ class. A pair of WSDL message elements and a pair of
1528 XML schema elements are generated for each WSDL operation. SOAP binding and address information
1529 is also generated. For example, a C++ header such as:

1530

```

1531 class MyService
1532 {
1533     public:
1534         int myMethod(std::string data);
1535         double myOtherMethod(double otherData);
1536 };

```

1537

1538 would generate WSDL like:

1539

```

1540 <?xml version="1.0" encoding="UTF-8"?>
1541 <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
1542     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
1543     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
1544     xmlns:tns="http://MyService"
1545     targetNamespace="http://MyService">
1546     <types>
1547         <xsd:schema targetNamespace="http://MyService"
1548             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
1549             xmlns:tns="http://MyService" elementFormDefault="qualified">
1550
1551             <xsd:element name="myMethod">
1552                 <xsd:complexType>
1553                     <xsd:sequence>
1554                         <xsd:element name="data" type="xsd:string"/>
1555                     </xsd:sequence>
1556                 </xsd:complexType>
1557             </xsd:element>
1558             <xsd:element name="myMethodResponse">
1559                 <xsd:complexType>
1560                     <xsd:sequence>
1561                         <xsd:element name="myMethodResponseData" type="xsd:short"/>
1562                     </xsd:sequence>
1563                 </xsd:complexType>
1564             </xsd:element>
1565             <xsd:element name="myOtherMethod">
1566                 <xsd:complexType>
1567                     <xsd:sequence>
1568                         <xsd:element name="otherData" type="xsd:double"/>
1569                     </xsd:sequence>
1570                 </xsd:complexType>
1571             </xsd:element>
1572             <xsd:element name="myOtherMethodResponse">
1573                 <xsd:complexType>
1574                     <xsd:sequence>
1575                         <xsd:element name="myOtherMethodResponseData" type="xsd:double"/>
1576                     </xsd:sequence>
1577                 </xsd:complexType>
1578             </xsd:element>
1579         </xsd:schema>
1580     </types>
1581     <message name="myMethodRequestMsg">
1582         <part name="body" element="tns:myMethod"/>
1583     </message>
1584     <message name="myMethodResponseMsg">
1585         <part name="body" element="tns:myMethodResponse"/>
1586     </message>
1587     <message name="myOtherMethodRequestMsg">
1588         <part name="body" element="tns:myOtherMethod"/>
1589     </message>
1590     <message name="myOtherMethodResponseMsg">
1591         <part name="body" element="tns:myOtherMethodResponse"/>
1592     </message>

```

```

1593 </message>
1594
1595 <portType name="MyServicePortType">
1596   <operation name="myMethod">
1597     <input message="tns:myMethodRequestMsg"/>
1598     <output message="tns:myMethodResponseMsg"/>
1599   </operation>
1600   <operation name="myOtherMethod">
1601     <input message="tns:myOtherMethodRequestMsg"/>
1602     <output message="tns:myOtherMethodResponseMsg"/>
1603   </operation>
1604 </portType>
1605
1606 <binding name="MyServiceBinding" type="tns:MyService">
1607   <soap:binding style="document"
1608     transport="http://schemas.xmlsoap.org/soap/http"/>
1609   <operation name="myMethod">
1610     <soap:operation soapAction="MyService#myMethod"/>
1611     <input>
1612       <soap:body use="literal"/>
1613     </input>
1614     <output>
1615       <soap:body use="literal"/>
1616     </output>
1617   </operation>
1618   <operation name="myOtherMethod">
1619     <soap:operation soapAction="MyService#myOtherMethod"/>
1620     <input>
1621       <soap:body use="literal"/>
1622     </input>
1623     <output>
1624       <soap:body use="literal"/>
1625     </output>
1626   </operation>
1627 </binding>
1628
1629 <service name="MyService">
1630   <port name="MyServicePort" binding="tns:MyServiceBinding">
1631     <soap:address location="http://server:9090/MyService"/>
1632   </port>
1633 </service>
1634 </definitions>

```

1635
1636 If multiple classes are defined in the single C++ header file, the class to be mapped must be specified by
1637 name.

1638 **12.4 Method mapping**

1639 **12.4.1 Default parameter value mapping**

1640 Where default values are defined in the parameters of a method, these are reflected in the schema as
1641 non-required elements. Default values in C++ method prototypes are generally provided to allow users to
1642 ignore the parameters.

1643
1644 E.g. a method prototype:

```

1645
1646 long myMethod(char* name, int id = 0, double value = 12.34);
1647

```

1648 would generate a schema like:

1649
1650
1651
1652
1653
1654
1655
1656
1657
1658

```
<xsd:element name="myMethod">  
  <xsd:complexType>  
    <xsd:sequence>  
      <xsd:element name="name" type="xsd:string"/>  
      <xsd:element name="id" type="xsd:short" minOccurs="0"/>  
      <xsd:element name="value" type="xsd:double" minOccurs="0"/>  
    </xsd:sequence>  
  </xsd:complexType>  
</xsd:element>
```

1659 **12.4.2 Non-named parameters and the return type**

1660 Above, we have seen method prototypes with named parameters. C++ allows prototype parameters to be
1661 unnamed, simply typed (e.g. long myMethod(char*, int, double)). Prototypes defined in this way are not
1662 supported.

1663

1664 The return type in C++ methods is unnamed, so, as has been shown above, an implementation MUST
1665 generate a name for the elements required by doc-lit-wrapped WSDL. E.g. for the method prototype
1666 above, the response data will be returned using the following schema:

1667

1668
1669
1670
1671
1672
1673
1674

```
<xsd:element name="myMethodResponse">  
  <xsd:complexType>  
    <xsd:sequence>  
      <xsd:element name="myMethodResponseData" type="xsd:short"/>  
    </xsd:sequence>  
  </xsd:complexType>  
</xsd:element>
```

1675 **12.4.3 The void return type**

1676 Handling of the void return type is controlled by the oneWay annotation. If oneWay is true, the operation
1677 will be mapped to a one-way (in-only) WSDL operation, otherwise it will be mapped to a request-response
1678 WSDL operation where the output message is empty.

1679

1680

```
void myMethod(char* name, double value);
```

1681

1682 would generate a schema like:

1683

1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695

```
<xsd:element name="myMethodRequestMsg">  
  <xsd:complexType>  
    <xsd:sequence>  
      <xsd:element name="name" type="xsd:string"/>  
      <xsd:element name="value" type="xsd:double"/>  
    </xsd:sequence>  
  </xsd:complexType>  
</xsd:element>  
  
<xsd:element name="myMethodResponseMsg">  
  <xsd:complexType/>  
</xsd:element>
```

1696

1697 and a WSDL operation in the WSDL portType and binding elements such as:

1698

1699
1700

```
<portType name="MyServicePortType">  
  <operation name="myMethod">
```

```

1701     <input message="tns:myMethodRequestMsg"/>
1702     <output message="tns:myMethodResponseMsg"/>
1703   </operation>
1704 </portType>
1705
1706 <binding name="MyServiceBinding" type="tns:MyService">
1707   <soap:binding style="document"
1708     transport="http://schemas.xmlsoap.org/soap/http"/>
1709   <operation name="myMethod">
1710     <soap:operation soapAction="MyService#myMethod"/>
1711     <input>
1712       <soap:body use="literal"/>
1713     </input>
1714     <output>
1715       <soap:body use="literal"/>
1716     </output>
1717   </operation>
1718 </binding>

```

1719

1720 Alternatively, if the `oneWay` attribute is specified on the method:

1721

```

1722 <interface.cpp header="LoanService.h">
1723   <method name="myMethod" oneWay="true" />
1724 </interface.cpp>

```

1725

1726 the following schema would be generated:

1727

```

1728 <xsd:element name="myMethodRequestMsg">
1729   <xsd:complexType>
1730     <xsd:sequence>
1731       <xsd:element name="name" type="xsd:string"/>
1732       <xsd:element name="value" type="xsd:double"/>
1733     </xsd:sequence>
1734   </xsd:complexType>
1735 </xsd:element>

```

1736

1737 and a WSDL operation in the WSDL `portType` and binding elements that contains no output element,
1738 such as:

1739

```

1740 <portType name="MyServicePortType">
1741   <operation name="myMethod">
1742     <input message="tns:myMethodRequestMsg"/>
1743   </operation>
1744 </portType>
1745
1746 <binding name="MyServiceBinding" type="tns:MyService">
1747   <soap:binding style="document"
1748     transport="http://schemas.xmlsoap.org/soap/http"/>
1749   <operation name="myMethod">
1750     <soap:operation soapAction="MyService#myMethod"/>
1751     <input>
1752       <soap:body use="literal"/>
1753     </input>
1754   </operation>
1755 </binding>

```


1756 **12.4.4 No Parameters Specified**

1757 If a C++ method prototype has no parameters, the input schema element is still required (for doc-lit-
1758 wrapped WSDL) but is empty. E.g. a method prototype:

1759

```
1760 int getValue();
```

1761

1762 would generate a schema like:

1763

```
1764 <xsd:element name="getValue">  
1765   <xsd:complexType/>  
1766 </xsd:element>  
1767  
1768 <xsd:element name="getValueResponse">  
1769   <xsd:complexType>  
1770     <xsd:sequence>  
1771       <xsd:element name="getValueResponseData" type="xsd:short"/>  
1772     </xsd:sequence>  
1773   </xsd:complexType>  
1774 </xsd:element>
```

1775 **12.4.5 In/Out Parameters**

1776 In/Out parameters allow the method to receive and change the value of a parameter with those changes
1777 being subsequently available in the invoking code. In/Out parameter are not needed for remotable calls
1778 so are not supported in this mapping.

1779 **12.4.6 Public Methods**

1780 All public methods of a C++ header will be converted to WSDL operation definitions.

1781 **12.4.7 Inherited Public Methods**

1782 Public methods inherited by a C++ class will not be converted to WSDL operation definitions. If an
1783 inherited method is required, it must be re-specified in the inheriting class.

1784 **12.4.8 Protected/Private Methods**

1785 Protected and private methods will not be converted to WSDL operation definitions.

1786 **12.4.9 Constructors/Destructors**

1787 Constructors and destructors will not be converted to WSDL operation definitions. The lack of state in
1788 standard web services makes explicit construction/destruction operations meaningless.

1789 **12.4.10 Overloaded Methods**

1790 Overloaded methods are not supported due to the lack of support for overloading in WSDL 1.

1791 **12.4.11 Operator overloading**

1792 Overloaded operators ("==", ">=", "new", etc) are not supported.

1793 **12.4.12 Exceptions**

1794 C++ method prototypes can specify that particular exceptions may be thrown by the method. Handling of
1795 C++ exception throw specifications is not defined by this mapping, and is implementation dependent.

1797 A C++ Annotations

1798 To allow developers to define SCA related information directly in source files, without having to separately
1799 author SCDL files, a set of annotations are defined. An SCA implementation MAY support these
1800 annotations. If annotations are supported by an implementation, the annotations defined here MUST be
1801 supported and MUST be mapped to SCDL as described. The SCA runtime MUST only process the
1802 SCDL files and not the annotations.

1803
1804 The annotations are defined as C++ comments in interface and implementation header files, for example:

```
1805  
1806 // @Scope("stateless")
```

1807 A.1 Application of Annotations to C++ Program Elements

1808 In general an annotation immediately precedes the program element it applies to. If multiple annotations
1809 apply to a program element, all of the annotations SHOULD be in the same comment block.

- 1810 • Class
1811 The annotation immediately precedes the class.

1812 Example

```
1813 // @Scope("composite")  
1814 class LoanServiceImpl : public LoanService {  
1815     ...  
1816 };
```

- 1817 • Method
1818 The annotation immediately precedes the method.

1819 Example:

```
1820 class LoanService  
1821 {  
1822     public:  
1823     // @OneWay  
1824     virtual void reportEvent(int eventId) = 0;  
1825     ...  
1826 };
```

- 1827 • Data Member
1828 The annotation immediately precedes the data member.

1829 Example:

```
1830 // @Property(name="loanType", type="xsd:int")  
1831 long loanType;
```

1832
1833 Annotations follow normal inheritance rules. An annotation on a base class or any element of a base
1834 class applies to any classes derived from the base class.

1835 A.2 Interface Header Annotations

1836 This section lists the annotations that may be used in the header file that defines a service interface.

1837 A.2.1 @Interface

1838 Annotation used to indicate a class defines an when multiple classes exist in a header file.

1839
1840 **Corresponds to:** class attribute of interface.cpp element.

1841
1842 **Format:**

```
1843 // @Interface
```

1844
1845 **Applies to:** Class

1847 **Example**

1848 **Interface header:**

```
1849 // @Interface  
1850 class LoanService {  
1851 ...  
1852 };
```

1853
1854 **Service definition:**

```
1855 <service name="LoanService">  
1856 <interface.cpp header="LoanService.h" class="LoanService" />  
1857 </service>
```

1858 **A.2.2 @Remotable**

1859 Annotation on service interface class to indicate that a service is remotable.

1860
1861 **Corresponds to:** remotable="true" attribute of interface.cpp element.

1862
1863 **Format:**

```
1864 // @Remotable
```

1865 The default is *false* (not remotable).

1866
1867 **Applies to:** Class

1868
1869 **Example**

1870 **Interface header:**

```
1871 // @Remotable  
1872 class LoanService {  
1873 ...  
1874 };
```

1875
1876 **Service definition:**

```
1877 <service name="LoanService">  
1878 <interface.cpp header="LoanService.h" remotable="true" />  
1879 </service>
```

1880 **A.2.3 @Callback**

1881 Annotation on a service interface class to specify the callback interface.

1882
1883 **Corresponds to:** callbackHeader and callbackClass attributes of interface.cpp element.

1884

1885 **Format:**

```
1886 // @Callback(header="headerName", class="className")
```

1887 where *headerName* is the name of the header defining the callback service interface. *className* is the
1888 optional name of the class for the callback interface.

1889

1890 **Applies to:** Class

1891

1892 **Example**

1893 **Interface header:**

```
1894 // @Callback(header="MyServiceCallback.h", class="MyServiceCallback")
1895 class MyService {
1896 public:
1897     virtual void someMethod( unsigned int arg ) = 0;
1898 };
```

1899

1900 **Service definition:**

```
1901 <service name="MyService">
1902     <interface.cpp header="MyService.h"
1903         callbackHeader="MyServiceCallback.h"
1904         callbackClass="MyServiceCallback" />
1905 </service>
```

1906 **A.2.4 @OneWay**

1907 Annotation on a service interface method to indicate the method is one way.

1908

1909 **Corresponds to:** `oneWay="true"` attribute of method element of an interface.cpp element.

1910

1911 **Format:**

```
1912 // @OneWay
```

1913

1914 **Applies to:** Method

1915

1916 **Example**

1917 **Interface header:**

```
1918 class LoanService
1919 {
1920 public:
1921     // @OneWay
1922     virtual void reportEvent(int eventId) = 0;
1923     ...
1924 };
```

1925

1926 **Service definition:**

```
1927 <service name="LoanService">
1928     <interface.cpp header="LoanService.h">
1929         <method name="reportEvent" oneWay="true" />
1930     </interface.cpp>
1931 </service>
```

1932 **A.2.5 @Conversational**

1933 Annotation on a service interface to denote a conversational service contract.

1934

1935 **Corresponds to:** requires="conversational" attribute of a service element.

1936

1937 **Format:**

```
1938 // @Conversational
```

1939

1940 **Applies to:** Class

1941

1942 **Example**

1943 **Interface header:**

```
1944 //@Conversational
1945 class LoanService
1946 {
1947     ...
1948 }
```

1949

1950 **Service definition:**

```
1951 <service name="LoanService" requires="conversational">
1952     <interface.cpp header="LoanService.h">
1953 </service>
```

1954 **A.2.6 @EndsConversation**

1955 Annotation on a service interface method to indicate that the conversation will be ended when this
1956 method is called

1957

1958 **Corresponds to:** endsConversation="true" attribute of method element of an interface.cpp element.

1959

1960 **Format:**

```
1961 // @EndsConversation
```

1962

1963 **Applies to:** Method

1964

1965 **Example**

1966 **Interface header:**

```
1967 class LoanService
1968 {
1969     public:
1970     // @EndsConversation
1971     virtual void cancelApplication( ) = 0;
1972     ...
1973 };
```

1974

1975 **Service definition:**

```
1976 <service name="LoanService">
1977     <interface.cpp header="LoanService.h">
1978         <method name="cancelApplication" endsConversation="true" />
```

1979 `</interface.cpp>`
1980 `</service>`

1981 **A.3 Implementation Header Annotations**

1982 This section lists the annotations that may be used in the header file that defines a service
1983 implementation.

1984 **A.3.1 @ComponentType**

1985 Annotation used to indicate which class implements a componentType when multiple classes exist in an
1986 implementation file.

1987

1988 **Corresponds to:** class attribute of implementation.cpp element.

1989

1990 **Format:**

1991 `// @ComponentType`

1992

1993 **Applies to:** Class

1994

1995 **Example**

1996 Implementation header:

1997 `// @ComponentType`
1998 `class LoanServiceImpl : public LoanService {`
1999 `...`
2000 `};`

2001

2002 Component definition:

2003 `<component name="LoanService">`
2004 `<implementation.cpp library="loan" header="LoanServiceImpl.h"`
2005 `class="LoanServiceImpl" />`
2006 `</component>`

2007 **A.3.2 @Scope**

2008 Annotation on a service implementation class to indicate the scope of the service.

2009

2010 **Corresponds to:** scope attribute of implementation.cpp element.

2011

2012 **Format:**

2013 `// @Scope("value")`

2014 where

2015 • **value** : [*stateless* |, *composite* |, *request* | *conversation*] (1..1) – specifies the scope of the
2016 implementation. The default value is *stateless*.

2017

2018 **Applies to:** Class

2019

2020 **Example**

2021 Implementation header:

```
2022 // @Scope("composite")
2023 class LoanServiceImpl : public LoanService {
2024     ...
2025 };
```

2026

2027 **Component definition:**

```
2028 <component name="LoanService">
2029     <implementation.cpp library="loan" header="LoanServiceImpl.h"
2030         scope="composite" />
2031 </component>
```

2032 **A.3.3 @EagerInit**

2033 Annotation on a service implementation class to indicate the implantation is to be instantiated when its
2034 containing component is started.

2035

2036 **Corresponds to:** eagerInit="true" attribute of implementation.cpp element.

2037

2038 **Format:**

```
2039 // @EagerInit
```

2040

2041 **Applies to:** Class

2042

2043 **Example**

2044 **Implementation header:**

```
2045 // @EagerInit
2046 class LoanServiceImpl : public LoanService {
2047     ...
2048 };
```

2049

2050 **Component definition:**

```
2051 <component name="LoanService">
2052     <implementation.cpp library="loan" header="LoanServiceImpl.h"
2053         eagerInit="true" />
2054 </component>
```

2055 **A.3.4 @AllowsPassByReference**

2056 Annotation on service implementation class or method to indicate that a service or method allows pass by
2057 reference semantics.

2058

2059 **Corresponds to:** allowsPassByReference="true" attribute of implementation.cpp element or a method
2060 element of an implementation.cpp element.

2061

2062 **Format:**

```
2063 // @AllowsPassByReference
```

2064

2065 **Applies to:** Class or Method

2066

2067 **Example**

2068 Implementation header:

```
2069 // @AllowsPassByReference
2070 class LoanService {
2071     ...
2072 };
```

2073

2074 Component definition:

```
2075 <component name="LoanService">
2076     <implementation.cpp library="loan" header="LoanServiceImpl.h"
2077         allowsPassByReference="true" />
2078 </component>
```

2079 A.3.5 @ConversationAttributes

2080 Annotation on a service implementation class to specify attributes of a conversational service.

2081

2082 **Corresponds to:** conversationMaxAge, conversationMaxIdle or conversationSinglePrincipal attributes of
2083 implementation.cpp element.

2084

2085 **Format:**

```
2086 // @ConversationAttributes(maxIdleTime="value", maxAge="value",
2087 singlePrincipal=boolValue)
```

2088 where

- 2089 • **maxIdleTime : string (0..1)** – specifies the maximum time that can pass between operations within a
2090 single conversation. **value** is a time expressed as an integer followed by a space and then one of the
2091 following: "seconds", "minutes", "hours", "days" or "years".
- 2092 • **maxAge : string (0..1)** – specifies the maximum time that the entire conversation can remain active.
2093 **value** is a time expressed as an integer followed by a space and then one of the following: "seconds",
2094 "minutes", "hours", "days" or "years".
- 2095 • **singlePrincipal : boolean (0..1)** – specifies if only the principal (the user) that started the
2096 conversation has authority to continue the conversation.

2097

2098 **Applies to:** Class

2099

2100 Example

2101 Implementation header:

```
2102 // @ConversationAttributes(maxAge="30 days", maxIdleTime="5 minutes",
2103 // singlePrincipal=false)
2104 class LoanServiceImpl : public LoanService
2105 {
2106     ...
2107 };
```

2108

2109 Component definition:

```
2110 <component name="LoanService">
2111     <implementation.cpp library="loan" header="LoanServiceImpl.h"
2112         conversationMaxAge="30 days" conversationMaxIdle="5 minutes"
2113         conversationSinglePrincipal="false" />
2114 </component>
```

2115 A.3.6 @Property

2116 Annotation on a service implementation class data member to define a property of the service.

2117

2118 **Corresponds to:** property element of componentType element.

2119

2120 **Format:**

```
2121 // @Property(name="propertyName", type="typeName"  
2122 //           default="defaultValue", required="true")
```

2123 where

- 2124 • **name : NCName (0..1)** - specifies the name of the property. If name is not specified the property
2125 name is taken from the name of the following data member.
- 2126 • **type : QName (0..1)** - specifies the type of the property. If not specified the type of the property is
2127 based on the C++ mapping of the type of the following data member to an xsd type as defined in [C++
2128 to WSDL Mapping](#). If the data member is an array, then the property is many-valued.
- 2129 • **required : boolean (0..1)** - specifies whether a value has to be set in the component definition for this
2130 property. Default is *false*
- 2131 • **default : <type> (0..1)** - specifies a default value and is only needed if *required* is *false*,

2132

2133 **Applies to:** DataMember

2134

2135 Example

2136 Implementation:

```
2137 // @Property(name="loanType", type="xsd:int")  
2138 long loanType;
```

2139

2140 Component Type definition:

```
2141 <componentType ... >  
2142   <service ... />  
2143   <property name="loanType" type="xsd:int" />  
2144 </componentType>
```

2145 A.3.7 @Reference

2146 Annotation on a service implementation class data member to define a reference of the service.

2147

2148 **Corresponds to:** reference element of componentType element.

2149

2150 **Format:**

```
2151 // @Reference(name="referenceName", interfaceHeader="LoanService.h",  
2152 //           interfaceClass="LoanService", required="true")
```

2153 where

- 2154 • **name : NCName (0..1)** - specifies the name of the reference. If name is not specified the reference
2155 name is taken from the name of the following data member.
- 2156 • **interfaceHeader : Name (1..1)** - specifies the C++ header defining the interface for the reference.
- 2157 • **interfaceClass : Name (0..1)** - specifies the C++ class defining the interface for the reference. If not
2158 specified the class is derived from the type of the annotated data member.
- 2159 • **required : boolean (0..1)** - specifies whether a value has to be set for this reference. Default is *true*.

2160

2161 If the annotated data member is a `std::list` then the implied component type has a reference with a
2162 multiplicity of either `0..n` or `1..n` depending on the value of the `@Reference` *required* attribute – `1..n`
2163 applies if `required=true`. Otherwise a multiplicity of `0..1` or `1..1` is implied.

2164

2165 **Applies to:** Data Member

2166

2167 **Example**

2168 **Implementation:**

```
2169 // @Reference(interfaceHeader="LoanService.h" required="true")  
2170 LoanService* loanService;  
2171  
2172 // @Reference(interfaceHeader="LoanService.h" required="false")  
2173 std::list<LoanService*> loanServices;
```

2174

2175 **Component Type definition:**

```
2176 <componentType ... >  
2177   <service ... />  
2178   <reference name="loanService" multiplicity="1..1">  
2179     <interface.cpp header="LoanService.h" class="LoanService" />  
2180   </reference>  
2181   <reference name="loanServices" multiplicity="0..n">  
2182     <interface.cpp header="LoanService.h" class="LoanService" />  
2183   </reference>  
2184 </componentType>
```

2185 B Policy Annotations for C++

2186 SCA provides facilities for the attachment of policy-related metadata to SCA assemblies, which influence
2187 how implementations, services and references behave at runtime. The policy facilities are described in
2188 **[POLICY]**. In particular, the facilities include Intents and Policy Sets, where intents express abstract,
2189 high-level policy requirements and policy sets express low-level detailed concrete policies.

2190

2191 Policy metadata can be added to SCA assemblies through the means of declarative statements placed
2192 into Composite documents and into Component Type documents. These annotations are completely
2193 independent of implementation code, allowing policy to be applied during the assembly and deployment
2194 phases of application development.

2195

2196 However, it can be useful and more natural to attach policy metadata directly to the code of
2197 implementations. This is particularly important where the policies concerned are relied on by the code
2198 itself. An example of this from the Security domain is where the implementation code expects to run
2199 under a specific security Role and where any service operations invoked on the implementation must be
2200 authorized to ensure that the client has the correct rights to use the operations concerned. By annotating
2201 the code with appropriate policy metadata, the developer can rest assured that this metadata is not lost or
2202 forgotten during the assembly and deployment phases.

2203

2204 The SCA C++ policy annotations provide the capability for the developer to attach policy information to
2205 C++ implementation code. The annotations concerned first provide general facilities for attaching SCA
2206 Intents and Policy Sets to C++ code. Secondly, there are further specific annotations that deal with
2207 particular policy intents for certain policy domains such as Security.

2208 B.1 General Intent Annotations

2209 SCA provides the annotation **@Requires** for the attachment of any intent to a C++ class, to a C++
2210 interface or to elements within classes and interfaces such as methods and members.

2211

2212 The @Requires annotation can attach one or multiple intents in a single statement.

2213

2214 Each intent is expressed as a string. Intents are XML QNames, which consist of a Namespace URI
2215 followed by the name of the Intent. The precise form used is as follows:

2216

```
2217 "{ " + Namespace URI + "}" + intentname
```

2218

2219 Intents may be qualified, in which case the string consists of the base intent name, followed by a ".",
2220 followed by the name of the qualifier. There may also be multiple levels of qualification.

2221

2222 This representation is quite verbose, so we expect that reusable constants will be defined for the
2223 namespace part of this string, as well as for each intent that is used by C++ code. SCA defines constants
2224 for intents such as the following:

2225

```
2226 #define SCA_PREFIX "{http://docs.oasis-open.org/ns/opencsa/sca/200712}"  
2227 #define CONFIDENTIALITY SCA_PREFIX ## "confidentiality"  
2228 #define CONFIDENTIALITY_MESSAGE CONFIDENTIALITY ## ".message"
```

2229

2230 Notice that, by convention, qualified intents include the qualifier as part of the name of the constant,
2231 separated by an underscore. These intent constants are defined in the file that defines an annotation for
2232 the intent (annotations for intents, and the formal definition of these constants, are covered in a following
2233 section).

2234

2235 Multiple intents (qualified or not) are expressed as separate strings within an array declaration.

2236

2237 **Corresponds to:** requires attribute of a service, reference, operation or property element.

2238

2239 **Format:**

```
2240 // @Requires("qualifiedIntent" | {"qualifiedIntent" [, "qualifiedIntent"]})
```

2241 where

```
2242 qualifiedIntent ::= QName | QName.qualifier | QName.qualifier1.qualifier2
```

2243

2244 **Applies to:** Class, Method, Data Member

2245

2246 **Examples**

2247 Attaching the intents "confidentiality.message" and "integrity.message".

```
2248 // @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
```

2249

2250 A reference requiring support for confidentiality:

```
2251 class Foo {  
2252     ...  
2253     // @Requires(CONFIDENTIALITY)  
2254     // @Reference(interfaceHeader="SetBar.h")  
2255     void setBar(Bar* bar);  
2256     ...  
2257 }
```

2258

2259 Users may also choose to only use constants for the namespace part of the QName, so that they may
2260 add new intents without having to define new constants. In that case, this definition would instead look
2261 like this:

2262

```
2263 class Foo {  
2264     ...  
2265     // @Requires(SCA_PREFIX "confidentiality")  
2266     // @Reference(interfaceHeader="SetBar.h")  
2267     void setBar(Bar* bar);  
2268     ...  
2269 }
```

2270 B.2 Specific Intent Annotations

2271 In addition to the general intent annotation supplied by the @Requires annotation described above, there
2272 are C++ annotations that correspond to some specific policy intents.

2273

2274 The general form of these specific intent annotations is an annotation with a name derived from the name
2275 of the intent itself. If the intent is a qualified intent, qualifiers are supplied as an attribute to the annotation
2276 in the form of a string or an array of strings.

2277

2278 For example, the SCA confidentiality intent described in [General Intent Annotations](#) using the
2279 @Requires(CONFIDENTIALITY) intent can also be specified with the specific @Confidentiality intent
2280 annotation. The specific intent annotation for the "integrity" security intent is:

```
2281 // @Integrity
```

2282

2283 **Corresponds to:** requires="<Intent>" attribute of a service, reference, operation or property element.

2284

2285 **Format:**

```
2286 // @<Intent>[(qualifiers)]
```

2287 where Intent is an NCName that denotes a particular type of intent.

```
2288 Intent ::= NCName
```

```
2289 qualifiers ::= "qualifier" | {"qualifier" [, "qualifier" ] }
```

```
2290 qualifier ::= NCName | NCName/qualifier
```

2291

2292 **Applies to:** Class, Method, Data Member – but see specific intents for restrictions

2293

2294 **Example**

```
2295 // @Authentication( {"message", "transport"} )
```

2296 This annotation attaches the pair of qualified intents: "authentication.message" and
2297 "authentication.transport" (the sca: namespace is assumed in both of these cases – "http://docs.oasis-
2298 open.org/ns/opencsa/sca/200712").

2299 B.3 Application of Intent Annotations

2300 Where multiple intent annotations (general or specific) are applied to the same C++ element, they are
2301 additive in effect. An example of multiple policy annotations being used together follows:

2302

```
2303 // @Authentication
```

```
2304 // @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
```

2305

2306 In this case, the effective intents are "authentication", "confidentiality.message" and "integrity.message".

2307

2308 If an annotation is specified at both the class/interface level and the method or field level, then the method
2309 or field level annotation completely overrides the class level annotation of the same type.

2310

2311 The intent annotation can be applied either to classes or to class methods when adding annotated policy
2312 on SCA services. Applying an intent to the setter method in a reference injection approach allows intents
2313 to be defined at references.

2314 B.4 Inheritance and Intent Annotations

2315 The following example shows the inheritance relations of intents on classes, operations, and super
2316 classes.

2317

```
2318 // @Remotable
```

```
2319 // @Integrity("transport")
```

```
2320 // @Authentication
```

```
2321 class HelloService {
```

```
2322 public:
```

```
2323 // @Integrity
```

```

2324 // @Authentication("message")
2325     wchar_t* hello(wchar_t* message) {...}
2326
2327 // @Integrity
2328 // @Authentication("transport")
2329     wchar_t* helloThere() {...}
2330 }
2331
2332 // @Remotable
2333 // @Confidentiality("message")
2334 class HelloChildService : public HelloService {
2335 public:
2336 // @Confidentiality("transport")
2337     wchar_t* hello(wchar_t* message) {...}
2338 // @Authentication
2339     wchar_t* helloWorld(){...}
2340 }

```

2341 Example 1a. Usage example of annotated policy and inheritance.

2342

- 2343 • The effective intent annotation on the helloWorld method is Integrity("transport"), @Authentication,
- 2344 and @Confidentiality("message").
- 2345 • The effective intent annotation on the hello method of the HelloChildService is @Integrity("transport"),
- 2346 @Authentication, and @Confidentiality("transport"),
- 2347 • The effective intent annotation on the helloThere method of the HelloChildService is @Integrity and
- 2348 @Authentication("transport"), the same as in HelloService class.
- 2349 • The effective intent annotation on the hello method of the HelloService is @Integrity and
- 2350 @Authentication("message")

2351

2352 The listing below contains the equivalent declarative security interaction policy of the HelloService and
 2353 HelloChildService implementation corresponding to the C++ classes shown in Example 1a.

2354

```

2355 <?xml version="1.0" encoding="ASCII"?>
2356
2357 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2358     name="HelloServiceComposite" >
2359     <service name="HelloService" requires="integrity/transport
2360         authentication">
2361         ...
2362     </service>
2363     <service name="HelloChildService" requires="integrity/transport
2364         authentication confidentiality/message">
2365         ...
2366     </service>
2367     ...
2368
2369     <component name="HelloServiceComponent">*
2370         <implementation.cpp library="HelloService.dll"
2371             header="HelloServiceImpl.h"/>
2372         <operation name="hello" requires="integrity
2373             authentication/message"/>
2374         <operation name="helloThere" requires="integrity
2375             authentication/transport"/>
2376     </component>
2377     <component name="HelloChildServiceComponent">*
2378         <implementation.cpp library="HelloChildService.dll"
2379             header="HelloChildServiceImpl.h" />
2380         <operation name="hello" requires="confidentiality/transport"/>
2381         <operation name="helloThere" requires=" integrity/transport

```

```

2382         authentication"/>
2383         <operation name="helloWorld" requires="authentication"/>
2384     </component>
2385
2386     ...
2387
2388 </composite>

```

2389 Example 1b. Declaratives intents equivalent to annotated intents in Example 1a.

2390 B.5 Relationship of Declarative and Annotated Intents

2391 Annotated intents on a C++ class cannot be overridden by declarative intents either in a composite
2392 document which uses the class as an implementation or by statements in a component Type document
2393 associated with the class. This rule follows the general rule for intents that they represent fundamental
2394 requirements of an implementation.

2395

2396 a declarative intent in a using composite document.

2397 B.6 Policy Set Annotations

2398 The SCA Policy Framework uses Policy Sets to capture detailed low-level concrete policies (for example,
2399 a concrete policy is the specific encryption algorithm to use when encrypting messages when using a
2400 specific communication protocol to link a reference to a service).

2401

2402 Policy Sets can be applied directly to C++ implementations using the **@PolicySets** annotation. The
2403 PolicySets annotation either takes the QName of a single policy set as a string or the name of two or
2404 more policy sets as an array of strings.

2405

2406 **Corresponds to:** policySets attribute of a service, reference, operation or property element.

2407

2408 **Format:**

```

2409 // @PolicySets( "<policy set QName>" |
2410 { "<policy set QName>" [, "<policy set QName>"] })

```

2411 As for intents, PolicySet names are QNames – in the form of "{Namespace-URI}localPart".

2412

2413 **Applies to:** Class, Method, Data Member

2414

2415 **Example**

```

2416 // @Reference(name="helloService", interfaceHeader="helloService.h",
2417 //           required=true)
2418 // @PolicySets({ MY_NS "WS_Encryption_Policy",
2419 //             MY_NS "WS_Authentication_Policy"})
2420 HelloService* helloService;
2421 ...

```

2422

2423 In this case, the Policy Sets WS_Encryption_Policy and WS_Authentication_Policy are applied, both
2424 using the namespace defined for the constant MY_NS.

2425

2426 PolicySets must satisfy intents expressed for the implementation when both are present, according to the
2427 rules defined in **[POLICY]**.

2428 **B.7 Security Policy Annotations**

2429 This section introduces annotations for SCA's security intents, as defined in **[POLICY]**.

2430 **B.7.1 Security Interaction Policy**

2431 The following interaction policy Intents and qualifiers are defined for Security Policy, which apply to the
2432 operation of services and references of an implementation:

- 2433 • @Integrity
- 2434 • @Confidentiality
- 2435 • @Authentication

2436

2437 All three of these intents have the same pair of Qualifiers:

- 2438 • message
- 2439 • transport

2440

2441 The following example shows an example of applying an intent to a reference. Accessing the hello
2442 operation of the referenced HelloService requires both "integrity.message" and "authentication.message"
2443 intents to be honored.

2444

```
2445 //Interface for HelloService
2446 class HelloService {
2447 public:
2448     virtual wchar_t* hello(wchar_t* helloMsg);
2449 }
2450
2451 // Interface for ClientService
2452 class ClientService {
2453 public:
2454     virtual void clientMethod();
2455 }
2456
2457 // Implementation class for ClientService
2458 #include "ComponentContext.h"
2459 #include "HelloService.h"
2460
2461 class ClientServiceImpl : public ClientService {
2462
2463 private:
2464     // @Reference(name="helloService", interfaceHeader="HelloService.h")
2465     // @Integrity("message")
2466     // @Authentication("message")
2467     HelloService* helloService;
2468
2469 public:
2470     void clientMethod() {
2471         ComponentContextPtr context = ComponentContext::getCurrent();
2472
2473         helloService = (HelloService* )context->getService("helloService");
2474
2475         wchar_t* result = helloService->hello(L"Hello World!");
2476
2477     }
2478 }
2479 }
```

2480 B.7.2 Security Implementation Policy

2481 SCA defines a number of security policy annotations that apply as policies to implementations
2482 themselves. These annotations mostly have to do with authorization and security identity. The following
2483 authorization and security identity annotations are supported:

- 2484 • @RunAs

2485 Takes as a parameter a string which is the name of a Security role, e.g. @RunAs("Manager").

2486 Code marked with this annotation will execute with the Security permissions of the identified role.

- 2487 • @RolesAllowed

2488 Takes as a parameter a single string or an array of strings which represent one or more role names.

2489 When present, the implementation can only be accessed by principals whose role corresponds to one
2490 of the role names listed in the @DeclareRoles attribute),

2491 e.g. @RolesAllowed({"Manager", "Employee"}).

2492 How role names are mapped to security principals is implementation dependent (SCA does not
2493 define this

- 2494 • @PermitAll

2495 No parameters. When present, grants access to all roles.

- 2496 • @DenyAll

2497 No parameters. When present, denies access to all roles.

- 2498 • @DeclareRoles

2499 Takes as a parameter a string or an array of strings which identify one or more role names that form
2500 the set of roles used by the implementation,

2501 e.g. @DeclareRoles({"Manager", "Employee", "Customer"})

2502

2503 For a full explanation of these intents, see [POLICY].

2504 B.7.2.1 Annotated Implementation Policy Example

2505 The following is an example showing annotated security implementation policy:

2506

```
2507 // @Remotable  
2508 class AccountService {  
2509 public:  
2510     virtual AccountReport* getAccountReport(char* customerID);  
2511 }
```

2512

2513 The following is a full listing of the AccountServiceImpl class, showing the Service it implements, plus the
2514 service references it makes and the settable properties that it has, along with a set of implementation
2515 policy annotations:

2516

```
2517 #include "commonj.sdo.h";  
2518 #include "AccountDataService.h";  
2519 #include "CheckingAccount.h";  
2520 #include "SavingsAccount.h";  
2521 #include "StockAccount.h";  
2522 #include "StockQuoteService.h";  
2523  
2524 // @RolesAllowed("customers")  
2525 // @RunAs("accountants" )  
2526 class AccountServiceImpl : public AccountService {
```

```

2527 protected:
2528 // @Property
2529     std::string currency = "USD";
2530
2531 // @Reference (name="accountDataService",
2532 //             interfaceHeader="AccountDataService.h")
2533     AccountDataService accountDataService;
2534
2535 // @Reference (name="stockQuoteService",
2536 //             interfaceHeader="StockQuoteService.h")
2537     StockQuoteService stockQuoteService;
2538
2539 public:
2540 // @RolesAllowed({"customers", "accountants"})
2541     AccountReport getAccountReport(char* customerID) {
2542
2543         DataFactory dataFactory = DataFactory::getDataFactory();
2544         AccountReport accountReport =
2545             (AccountReport) dataFactory ->create (AccountReportType);
2546         DataObjectList accountSummaries = accountReport->getAccountSummaries();
2547
2548         CheckingAccount checkingAccount =
2549             accountDataService->getCheckingAccount (customerID);
2550         AccountSummary checkingAccountSummary =
2551             (AccountSummary) dataFactory ->create (AccountSummaryType);
2552         checkingAccountSummary->setAccountNumber (
2553             checkingAccount->getAccountNumber ());
2554         checkingAccountSummary->setAccountType ("checking");
2555         checkingAccountSummary->setBalance (fromUSDollarToCurrency (
2556             checkingAccount->getBalance ());
2557         accountSummaries->append (checkingAccountSummary);
2558
2559         SavingsAccount savingsAccount =
2560             accountDataService->getSavingsAccount (customerID);
2561         AccountSummary savingsAccountSummary =
2562             (AccountSummary) dataFactory->create (AccountSummaryType);
2563         savingsAccountSummary->setAccountNumber (
2564             savingsAccount->getAccountNumber ());
2565         savingsAccountSummary->setAccountType ("savings");
2566         savingsAccountSummary->setBalance (fromUSDollarToCurrency (
2567             savingsAccount->getBalance ());
2568         accountSummaries->append (savingsAccountSummary);
2569
2570         StockAccount stockAccount =
2571             accountDataService->getStockAccount (customerID);
2572         AccountSummary stockAccountSummary =
2573             (AccountSummary) dataFactory->create (AccountSummaryType);
2574         stockAccountSummary->setAccountNumber (stockAccount->getAccountNumber ());
2575         stockAccountSummary->setAccountType ("stock");
2576         float balance = (stockQuoteService->getQuote (
2577             stockAccount->getSymbol ()) *
2578             stockAccount->getQuantity ());
2579         stockAccountSummary->setBalance (fromUSDollarToCurrency (balance));
2580         accountSummaries->append (stockAccountSummary);
2581
2582         return accountReport;
2583     }
2584
2585 // @PermitAll
2586     float fromUSDollarToCurrency(float value){
2587
2588         if (currency == "USD")
2589             return value;
2590         else if (currency == "EURO")

```

```
2591     return value * 0.8;
2592     else
2593     return 0.0;
2594     }
2595 }
```

2596

2597 In this example, the implementation class as a whole is marked:

- 2598 • `@RolesAllowed("customers")` - indicating that customers have access to the implementation as a
2599 whole
- 2600 • `@RunAs("accountants")` – indicating that the code in the implementation runs with the permissions of
2601 accountants

2602

2603 The `getAccountReport(..)` method is marked with `@RolesAllowed({"customers", "accountants"})`, which
2604 indicates that this method can be called by both customers and accountants.

2605

2606 The `fromUSDollarToCurrency()` method is marked with `@PermitAll`, which means that this method can be
2607 called by any role.

2608

C XML Schemas

2609 Two XML schemas are defined to support the use of C++ for implementation and definition of interfaces.

2610 The normative schemas are located at:

2611 <http://docs.oasis-open.org/opencsa/sca-c-cpp/sca-interface-cpp-1.1-schema.xsd>

2612 and

2613 <http://docs.oasis-open.org/opencsa/sca-c-cpp/sca-implementation-cpp-1.1-schema.xsd>

2614

2615 The following copies are provided for reference.

2616 C.1 sca-interface-cpp-1.1-schema.xsd

```
2617 <?xml version="1.0" encoding="UTF-8"?>
2618 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2619         targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2620         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2621         xmlns:sdo="commonj.sdo/XML"
2622         elementFormDefault="qualified">
2623
2624     <include schemaLocation="sca-core.xsd"/>
2625
2626     <element name="interface.cpp" type="sca:CPPInterface"
2627             substitutionGroup="sca:interface"/>
2628
2629     <complexType name="CPPInterface">
2630         <complexContent>
2631             <extension base="sca:Interface">
2632                 <sequence>
2633                     <element name="method" type="sca:CPPMethod"
2634                             minOccurs="0" maxOccurs="unbounded" />
2635                     <any namespace="##other" processContents="lax"
2636                             minOccurs="0" maxOccurs="unbounded"/>
2637                 </sequence>
2638                 <attribute name="header" type="NCName" use="required"/>
2639                 <attribute name="class" type="Name" use="required"/>
2640                 <attribute name="callbackHeader" type="NCName" use="optional"/>
2641                 <attribute name="callbackClass" type="Name" use="optional"/>
2642                 <attribute name="remotable" type="boolean" use="optional"/>
2643                 <anyAttribute namespace="##other" processContents="lax"/>
2644             </extension>
2645         </complexContent>
2646     </complexType>
2647
2648     <complexType name="CPPMethod">
2649         <attribute name="name" type="NCName" use="required"/>
2650         <attribute name="oneWay" type="boolean" use="optional"/>
2651         <attribute name="endsConversation" type="boolean" use="optional"/>
2652         <anyAttribute namespace="##other" processContents="lax"/>
2653     </complexType>
2654
2655 </schema>
```

2656 C.2 sca-implementation-cpp-1.1-schema.xsd

```
2657 <?xml version="1.0" encoding="UTF-8"?>
2658 <schema xmlns="http://www.w3.org/2001/XMLSchema"
```

```

2659     targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2660     xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2661     xmlns:sdo="commonj.sdo/XML"
2662     elementFormDefault="qualified">
2663
2664     <include schemaLocation="sca-core.xsd"/>
2665
2666     <element name="implementation.cpp" type="sca:CPPImplementation"
2667         substitutionGroup="sca:implementation" />
2668     <complexType name="CPPImplementation">
2669         <complexContent>
2670             <extension base="sca:Implementation">
2671                 <sequence>
2672                     <element name="method" type="sca:CPPImplementationMethod"
2673                         minOccurs="0" maxOccurs="unbounded" />
2674                     <any namespace="##other" processContents="lax"
2675                         minOccurs="0" maxOccurs="unbounded"/>
2676                 </sequence>
2677                 <attribute name="library" type="NCName" use="required"/>
2678                 <attribute name="header" type="NCName" use="required"/>
2679                 <attribute name="path" type="Name" use="optional"/>
2680                 <attribute name="class" type="Name" use="optional"/>
2681                 <attribute name="componentType" type="NCName" use="optional"/>
2682                 <attribute name="scope" type="sca:CPPImplementationScope"
2683                     use="optional"/>
2684                 <attribute name="eagerInit" type="boolean" use="optional"/>
2685                 <attribute name="allowsPassByReference" type="boolean"
2686                     use="optional"/>
2687                 <attribute name="conversationMaxAge" type="string"
2688                     use="optional"/>
2689                 <attribute name="conversationMaxIdle" type="string"
2690                     use="optional"/>
2691                 <attribute name="conversationSinglePrincipal" type="boolean"
2692                     use="optional"/>
2693                 <anyAttribute namespace="##other" processContents="lax"/>
2694             </extension>
2695         </complexContent>
2696     </complexType>
2697
2698     <simpleType name="CPPImplementationScope">
2699         <restriction base="string">
2700             <enumeration value="stateless"/>
2701             <enumeration value="composite"/>
2702             <enumeration value="request"/>
2703             <enumeration value="converstion"/>
2704         </restriction>
2705     </simpleType>
2706
2707     <complexType name="CPPImplementationMethod">
2708         <attribute name="name" type="NCName" use="required"/>
2709         <attribute name="allowsPassByReference" type="boolean"
2710             use="optional"/>
2711         <anyAttribute namespace="##other" processContents="lax"/>
2712     </complexType>
2713
2714 </schema>

```

2715 **D Migration**

2716 To aid migration of an implementation or clients using an implementation based the version of the Service
2717 Component Architecture for C++ defined in [OSOA SCA C++ Client and Implementation V1.00](#), this
2718 appendix identifies the relevant changes to APIs, annotations, or behavior defined in V1.00.

2719 **D.1 Annotations related to conversations**

2720 @Conversation has been changed to @ConversationAttributes. @Conversation is removed.

2721 @EndConversation has been changed to @EndsConversation. @EndConversation is removed.

2722

E Acknowledgements

2723 The following individuals have participated in the creation of this specification and are gratefully
2724 acknowledged:

2725 **Participants:**

2726 Andrew Borley, IBM

2727 Bryan Aupperle, IBM

2728 David Haney, Rogue Wave Software

2729 Jeff Mischkin, Oracle

2730 Mike Edwards, IBM

2731 Pete Robbins, IBM

F Non-Normative Text

2733

G Revision History

2734 [optional; should not be included in OASIS Standards]

2735

Revision	Date	Editor	Changes Made
5	2008-3-17	Bryan Aupperle	<ul style="list-style-type: none"> Editorial changes in preparation for Committee Draft
4	2008-2-26	Bryan Aupperle	<ul style="list-style-type: none"> Remove duplicated text from Assembly Spec Reformat pseudo-schema presentation and description to be consistent with assembly spec Incorporate changes for CCPP-32, CCPP-34 and CCPP-35
3	2008-1-24	Bryan Aupperle	<ul style="list-style-type: none"> Conformance statement cleanup Incorporate changes for CCPP-21, CCPP-26 and CCPP-31
2	2007-11-02	David Haney, Bryan Aupperle	<ul style="list-style-type: none"> Add initial abstract (based on Introduction), updated Editors list, corrected typo in Acknowledgements. Incorporate changes proposed in CCPP-12, CCPP-13, CCPP-14. Update MUST/must, SHOULD/should and MAY/may to reflect TC use or RFC 2119 Incorporate changes for CCPP-1, CCPP-4, CCPP-10, CCPP-15 CCPP-18 and CCPP-19
1	2007-09-22	David Haney	Initial conversion of the OSOA 1.0 specification to the OASIS template.

2736