



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

Committee Draft 02

5 February 2009

Specification URIs:

This Version:

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

Previous Version:

<http://docs.oasis-open.org/opencsa/sca-c-cpp/sca-cppcni-1.1-spec-cd01.html>
<http://docs.oasis-open.org/opencsa/sca-c-cpp/sca-cppcni-1.1-spec-cd01.doc>
<http://docs.oasis-open.org/opencsa/sca-c-cpp/sca-cppcni-1.1-spec-cd01.pdf> (Authoritative)

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)

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>
<http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901>

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 operations.

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 operations.

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® 2006, 2009. 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	8
1.1	Terminology	8
1.2	Normative References	8
1.3	Non-Normative References	9
1.4	Conventions	9
1.4.1	Naming Conventions	9
1.4.2	Typographic Conventions	9
2	Basic Component Implementation Model	10
2.1	Implementing a Service	10
2.1.1	Implementing a Remotable Service	11
2.1.2	Implementing a Local Service	12
2.2	Component Implementation Scopes	12
2.2.1	Stateless scope	12
2.2.2	Composite scope	13
2.3	Implementing a Configuration Property	13
2.4	Component Type and Component	13
2.4.1	Interface.cpp	15
2.4.2	Function and CallbackFunction	15
2.4.3	Implementation.cpp	16
2.4.4	Implementation Function	17
2.5	Instantiation	17
3	Basic Client Model	18
3.1	Accessing Services from Component Implementations	18
3.2	Accessing Services from non-SCA component implementations	19
3.3	Calling Service Operations	19
4	Asynchronous Programming	20
4.1	Non-blocking Calls	20
4.2	Callbacks	20
4.2.1	Using Callbacks	21
4.2.2	Callback Instance Management	22
4.2.3	Implementing Multiple Bidirectional Interfaces	22
5	Error Handling	23
6	C++ API	24
6.1	Reference Counting Pointers	24
6.1.1	operator*	24
6.1.2	operator->	24
6.1.3	operator void*	25
6.1.4	operator!	25
6.2	Component Context	25
6.2.1	getCurrent	26
6.2.2	getURI	26
6.2.3	getService	26
6.2.4	getServices	26

6.2.5	getServiceReference	27
6.2.6	getServiceReferences	27
6.2.7	getProperties	27
6.2.8	getDataFactory	27
6.2.9	getSelfReference	28
6.3	ServiceReference	28
6.3.1	getService	28
6.3.2	getCallback	29
6.4	SCAException	29
6.4.1	getEClassName	29
6.4.2	getMessageText	29
6.4.3	getFileName	30
6.4.4	getLineNumber	30
6.4.5	getFunctionName	30
6.5	SCANullPointerException	30
6.6	ServiceRuntimeException	30
6.7	ServiceUnavailableException	31
6.8	MultipleServicesException	31
7	C++ Contributions	32
7.1	Executable files	32
7.1.1	Executable in contribution	32
7.1.2	Executable shared with other contribution(s) (Export)	32
7.1.3	Executable outside of contribution (Import)	33
7.2	componentType files	34
7.3	C++ Contribution Extensions	34
7.3.1	Export.cpp	34
7.3.2	Import.cpp	35
8	Types Supported in Service Interfaces	36
8.1	Local service	36
8.2	Remotable service	36
9	Restrictions on C++ header files	37
10	WSDL to C++ and C++ to WSDL Mapping	38
10.1	Augmentations for WSDL to C++ Mapping	39
10.1.1	Mapping WSDL targetNamespace to a C++ namespace	39
10.1.2	Mapping WSDL Faults to C++ Exceptions	39
10.1.3	Mapping of in, out, in/out parts to C++ member function parameters	39
10.2	Augmentations for C++ to WSDL Mapping	40
10.2.1	Mapping C++ namespaces to WSDL namespaces	40
10.2.2	Parameter and return type classification	40
10.2.3	C++ to WSDL Type Conversion	40
10.2.4	Service-specific Exceptions	40
10.3	SDO Data Binding	40
10.3.1	Simple Content Binding	40
10.3.2	Complex Content Binding	42
11	Conformance	43

11.1	Conformance Targets	43
11.2	Conformance Claims.....	43
11.3	SCA Implementations	43
11.4	SCDL Documents	43
11.5	C++ Component Implementations	43
11.6	C++ Header Files	43
11.7	WSDL Files	44
11.8	Extensions.....	44
A	C++ SCA Annotations.....	45
A.1	Application of Annotations to C++ Program Elements	45
A.2	Interface Header Annotations	45
A.2.1	@Interface	46
A.2.2	@Remotable	46
A.2.3	@Callback	47
A.2.4	@OneWay.....	47
A.3	Implementation Header Annotations	48
A.3.1	@ComponentType	48
A.3.2	@Scope.....	48
A.3.3	@EagerInit	49
A.3.4	@AllowsPassByReference	49
A.3.5	@Property	50
A.3.6	@Reference	51
A.4	Base Annotation Grammar	51
B	C++ SCA Policy Annotations	53
B.1	General Intent Annotations	53
B.2	Specific Intent Annotations	54
B.2.1	Security Interaction.....	55
B.2.2	Security Implementation.....	55
B.2.3	Reliable Messaging	56
B.2.4	Transactions.....	56
B.2.5	Miscellaneous.....	56
B.3	Application of Intent Annotations	56
B.4	Inheritance and Intent Annotations.....	57
B.5	Relationship of Declarative and Annotated Intents	58
B.6	Policy Set Annotations.....	58
B.7	Policy Annotation Grammar Additions	59
B.8	Annotation Constants	59
C	C++ WSDL Mapping Annotations.....	60
C.1	Interface Header Annotations.....	60
C.1.1	@WebService	60
C.1.2	@WebFunction	61
C.1.3	@OneWay.....	63
C.1.4	@WebParam.....	64
C.1.5	@WebResult	66
C.1.6	@SOAPBinding.....	68

C.1.7 @WebFault	69
C.1.8 @WebThrows	71
D WSDL C++ Mapping Extensions	72
D.1 <cpp:bindings>	72
D.2 <cpp:class>	72
D.3 <cpp:enableWrapperStyle>	73
D.4 <cpp:namespace>	74
D.5 <cpp:memberFunction>	75
D.6 <cpp:parameter>	76
D.7 JAX-WS WSDL Extensions	78
D.8 WSDL Extensions Schema	79
E XML Schemas	81
E.1 sca-interface-cpp-1.1.xsd	81
E.2 sca-implementation-cpp-1.1.xsd	82
E.3 sca-contribution-cpp-1.1.xsd	82
F Conformance Items	84
F.1 JAX-WS Conformance	87
F.1.1 Ignored Conformance Statements	89
G Migration	91
G.1 Method child elements of interface.cpp and implementation.cpp	91
H Acknowledgements	92
I Non-Normative Text	93
J Revision History	94

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 operations.

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 operations.

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]

This specification uses predefined namespace prefixes throughout; they are given in the following list. Note that the choice of any namespace prefix is arbitrary and not semantically significant.

Table 1-1 Prefixes and Namespaces used in this specification

Prefix	Namespace	Notes
xs	"http://www.w3.org/2001/XMLSchema"	Defined by XML Schema 1.0 specification
sca	"http://docs.oasis-open.org/ns/opencsa/sca/200712"	Defined by the SCA specifications
cpp	"http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"	

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 2009
- [POLICY] D. Booz, 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 2009
- [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

- 37 **[XSD]** P. Brion, et al., *XML Schema Part 2: Datatypes Second Edition*,
38 <http://www.w3.org/TR/xmlschema-2/>, W3C XML Schema Part 2: Datatypes
39 Second Edition, October 2004
- 40 **[JAXWS21]** D. Kohlert, et al., *The Java API for XML-Based Web Services (JAX-WS) 2.1*,
41 <http://jcp.org/aboutJava/communityprocess/mrel/jsr224/index2.html>, JCP Java
42 API for XML-Based Web Services (JAX-WS) 2.1, May 2007

43 **1.3 Non-Normative References**

44 **1.4 Conventions**

45 **1.4.1 Naming Conventions**

46 This specification follows some naming conventions for artifacts defined by the specification, as follows:

- 47 • For the names of elements and the names of attributes within XSD files, the names follow the
48 CamelCase convention, with all names starting with a lower case letter.
49 e.g. `<element name="componentType" type="sca:ComponentType"/>`
- 50 • For the names of types within XSD files, the names follow the CamelCase convention with all names
51 starting with an upper case letter
52 e.g. `<complexType name="ComponentService">`
- 53 • For the names of intents, the names follow the CamelCase convention, with all names starting with a
54 lower case letter, EXCEPT for cases where the intent represents an established acronym, in which
55 case the entire name is in upper case.
56 An example of an intent which is an acronym is the "SOAP" intent.

57 **1.4.2 Typographic Conventions**

58 This specification follows some typographic conventions for some specific constructs

- 59 • XML attributes are identified in text as `@attribute`
- 60 • Language identifiers used in text are in `courier`
- 61 • Literals in text are in *italics*

2 Basic Component Implementation Model

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.

A service provided by a C++ implementation has an interface (a **service interface**) which is defined using one of:

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

An abstract base class is a class which has only pure virtual member functions. A C++ implementation **MUST** implement all of the operation(s) of the service interface(s) of its componentType. [**CPP20001**]

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.

```
#include "LoanServiceImpl.h"
```

```

108 LoanServiceImpl::LoanServiceImpl ()
109 {
110     ...
111 }
112
113 LoanServiceImpl::~LoanServiceImpl ()
114 {
115     ...
116 }
117
118 bool LoanServiceImpl::approveLoan(unsigned long customerNumber,
119                                 unsigned long loanAmount)
120 {
121     ...
122 }

```

123

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

125

```

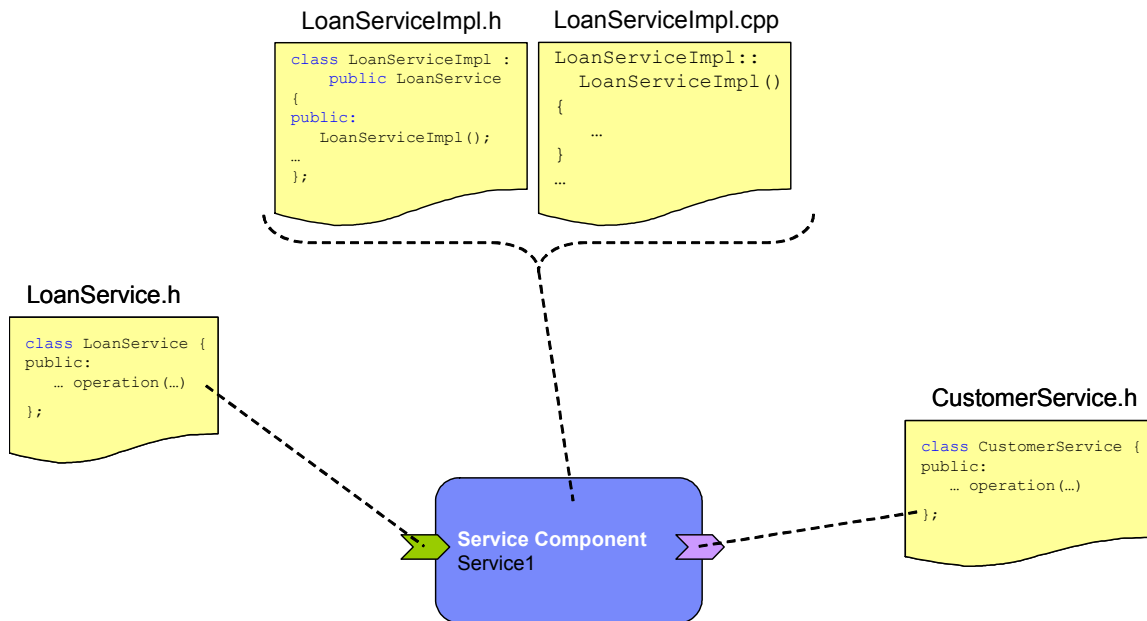
126 <?xml version="1.0" encoding="ASCII"?>
127 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
128   <service name="LoanService">
129     <interface.cpp header="LoanService.h"/>
130   </service>
131 </componentType>

```

132

133 The following picture shows the relationship between the C++ header files and implementation files for a
 134 component that has a single service and a single reference.
 135

135



136

137 2.1.1 Implementing a Remotable Service

138 A `@remotable="true"` attribute on an `interface.cpp` element indicates that the interface is **remotable** as
 139 described in the Assembly Specification [ASSEMBLY]. The following snippet shows the component type
 140 for a remotable service:

141

```

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

```

```
143 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
144   <service name="LoanService">
145     <interface.cpp header="LoanService.h" remotable="true"/>
146   </service>
147 </componentType>
```

148
149 Complex data types exchanged via remotable service interfaces MUST be compatible with the
150 marshalling technology that is used by the service binding.

151
152 An implementation of a remotable service can declare whether it allows pass by reference data exchange
153 semantics on calls to it, meaning that the by-value semantics can be maintained without requiring that the
154 parameters be copied. A C++ implementation of a remotable service that allows pass by reference MUST
155 NOT alter its input data during or after the invocation, and MUST NOT modify return data after invocation.
156 [CPP20002] The `@allowsPassByReference=true` attribute on the `implementation.cpp` element of a
157 remotable service is used to declare that calls to the whole interface allows pass by reference.
158 Alternatively, this attribute can be used on a specific member function.

159 2.1.2 Implementing a Local Service

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

161 2.2 Component Implementation Scopes

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

166
167 Scopes are specified using the `@scope` attribute of the `implementation.cpp` element.

168
169 When a scope is not specified on an implementation class, the SCA runtime will interpret the
170 implementation scope as **stateless**.

171
172 An SCA runtime MUST support these scopes; **stateless** and **composite**. Additional scopes MAY be
173 provided by SCA runtimes. [CPP20003]

174
175 The following snippet shows the component type for a composite scoped component:

```
176  
177 <component name="LoanService">
178   <implementation.cpp library="loan" class="LoanServiceImpl"
179     scope="composite"/>
180 </component>
```

181 2.2.1 Stateless scope

182 For stateless scope components, there is no implied correlation between implementation instances used
183 to dispatch service requests.

184
185 The concurrency model for the stateless scope is single threaded. An SCA runtime MUST ensure that a
186 stateless scoped implementation instance object is only ever dispatched on one thread at any one time.
187 In addition, within the SCA lifecycle of an instance, an SCA runtime MUST only make a single invocation
188 of one business method. [CPP20012]

189 2.2.2 Composite scope

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

193

194 A composite scoped implementation may also specify eager initialization using the `@eagerInit="true"`
195 attribute on the `implementation.cpp` element of a component definition. When marked for eager
196 initialization, the composite scoped instance will be created when its containing component is started.

197

198 The concurrency model for the composite scope is multi-threaded. An SCA runtime MAY run multiple
199 threads in a single composite scoped implementation instance object and it MUST NOT perform any
200 synchronization. [CPP20013]

201 2.3 Implementing a Configuration Property

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

205

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

207

```
208 #include "ComponentContext.h"  
209 using namespace oasis::sca;  
210  
211 void clientFunction()  
212 {  
213     ...  
214  
215     ComponentContext context = ComponentContext::getCurrent();  
216  
217     DataObjectPtr properties = context.getProperties();  
218  
219     long loanRating = properties->getInteger("maxLoanValue");  
220  
221     ...  
222 }
```

223 2.4 Component Type and Component

224 For a C++ component implementation, a component type is specified in a side file. By default, the
225 `componentType` side file is in the root directory of the composite containing the component or some
226 subdirectory of the composite root directory with a name matching the implementation class of the
227 component. The location can be modified as described below.

228

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

231

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

234

```
235 // LoanService interface  
236 class LoanService {  
237     public:
```

```
238     virtual bool approveLoan(unsigned long customerNumber,  
239                             unsigned long loanAmount) = 0;  
240 };
```

241

242 Implementation declaration header file.

243

```
244 class LoanServiceImpl : public LoanService {  
245 public:  
246     LoanServiceImpl();  
247     virtual ~LoanServiceImpl();  
248  
249     virtual bool approveLoan(unsigned long customerNumber,  
250                             unsigned long loanAmount);  
251 };
```

252

253 Implementation.

254

```
255 #include "LoanServiceImpl.h"  
256  
257 ///////////////////////////////////////////////////////////////////  
258 // Construction/Destruction  
259 ///////////////////////////////////////////////////////////////////  
260 LoanServiceImpl::LoanServiceImpl()  
261 {  
262     ...  
263 }  
264 LoanServiceImpl::~LoanServiceImpl()  
265 {  
266     ...  
267 }  
268 ///////////////////////////////////////////////////////////////////  
269 // Implementation  
270 ///////////////////////////////////////////////////////////////////  
271 bool LoanServiceImpl::approveLoan(unsigned long customerNumber,  
272                                   unsigned long loanAmount)  
273 {  
274     ...  
275 }
```

276

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

278

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

285

286 The following snippet shows the component using the implementation.

287

```
288 <?xml version="1.0" encoding="ASCII"?>  
289 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
290     name="LoanComposite" >  
291     ...  
292  
293     <component name="LoanService">  
294         <implementation.cpp library="loan" class="LoanServiceImpl"/>
```

```
295     </component>
296 </composite>
```

297 2.4.1 Interface.cpp

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

300

```
301 <?xml version="1.0" encoding="ASCII"?>
302 <!-- interface.cpp schema snippet -->
303 <interface.cpp xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
304     header="string" class="Name"? remotal="boolean"?
305     callbackHeader="string" callbackClass="Name"? >
306
307     <function ... />*
308     <callbackFunction ... />*
309
310 </interface.cpp>
```

311

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

- 313 • **header : string (1..1)** – full name of the header file that describes the interface, including relative path
314 from the composite root.
- 315 • **class : Name (0..1)** – name of the class declaration for the interface in the header file, including any
316 namespace definition. If the header file identified by the **@header** attribute of an **<interface.cpp>**
317 element contains more than one class, then the **@class** attribute **MUST** be specified for the
318 **<interface.cpp>** element. [CPP20005]
- 319 • **callbackHeader : string (0..1)** – full name of the header file that describes the callback interface,
320 including relative path from the composite root.
- 321 • **callbackClass : Name (0..1)** – name of the class declaration for the callback interface in the callback
322 header file, including any namespace definition. If the header file identified by the **@callbackHeader**
323 attribute of an **<interface.cpp>** element contains more than one class, then the **@callbackClass**
324 attribute **MUST** be specified for the **<interface.cpp>** element. [CPP20006]
- 325 • **remotal : boolean (0..1)** – indicates whether the service is remotable or local. The default is local.
326 See Implementing a Remotal Service

327

328 The **interface.cpp** element has the following **child elements**:

329 **function : CPPFunction (0..n)** – see Function and CallbackFunction

330 **callbackFunction : CPPFunction (0..n)** – see Function and CallbackFunction

331 2.4.2 Function and CallbackFunction

332 Some member functions of an interface have behavioral characteristics, which will be described later, that
333 need to be identified. This is done using a **function** or **callbackFunction** child element of **interface.cpp**

334

335 The following snippet shows the **interface.cpp** schema with the schema for the **function** and
336 **callbackFunction** child elements:

337

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

344
345 `</interface.cpp>`

346
347 The **function** and **callbackFunction** elements have the following **attributes**:

- 348 • **name : NCName (1..1)** – name of the method being decorated. The **@name** attribute of a `<function/>`
349 child element of a `<interface.cpp/>` MUST be unique amongst the `<function/>` elements of that
350 `<interface.cpp/>`. [CPP20007]
351 The **@name** attribute of a `<callbackFunction/>` child element of a `<interface.cpp/>` MUST be unique
352 amongst the `<callbackFunction/>` elements of that `<interface.cpp/>`. [CPP20008]
- 353 • **oneWay : boolean (0..1)** – see Non-blocking Calls

354 2.4.3 Implementation.cpp

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

```
357  
358 <?xml version="1.0" encoding="ASCII"?>  
359 <!-- implementation.cpp schema snippet -->  
360 <implementation.cpp xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
361     library="NCName" path="string"? class="Name"  
362     scope="scope"? componentType="string"? allowsPassByReference="Boolean"?  
363     eagerInit="boolean"? >  
364  
365     <method ... />*  
366  
367 </implementation.cpp>
```

368
369 The **implementation.cpp** element has the following **attributes**:

- 370 • **library : NCName (1..1)** – name of the dll or shared library that holds the factory for the service
371 component. This is the root name of the library.
- 372 • **path : string (0..1)** - path to the library which is either relative to the root of the contribution containing
373 the composite or is prefixed with a contribution import name and is relative to the root of the import.
374 See C++ Contributions.
- 375 • **class : Name (1..1)** – name of the class declaration of the implementation, including any namespace
376 definition. The name of the componentType file for a C++ implementation MUST match the class
377 name (excluding any namespace definition) of the implementations as defined by the **@class** attribute
378 of the `<implementation.cpp/>` element. [CPP20009] The SCA runtime will append `.componentType` to
379 the class name to find the componentType file.
- 380 • **scope : CPPImplementationScope (0..1)** – identifies the scope of the component implementation.
381 The default is stateless. See Component Implementation Scopes
- 382 • **componentType : string (0..1)** – path to the componentType file which is relative to the root of the
383 contribution containing the composite or is prefixed with a contribution import name and is relative to
384 the root of the import.
- 385 • **allowsPassByReference : boolean (0..1)** – indicates the service allows pass by reference data
386 exchange semantics on calls to it. See Implementing a Remotable Service
- 387 • **eagerInit type : boolean (0..1)** – indicates a composite scoped implementation should be initialized
388 when it is loaded. See Composite scope

389
390 The **implementation.cpp** element has the following **child element**:

- 391 **function : CPPImplementationMethod (0..n)** – see Implementation Function

392 2.4.4 Implementation Function

393 Some member functions of an implementation have operational characteristics that need to be identified.
394 This is done using a *function* child element of *implementation.cpp*

395

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

397

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

405

406 The *function* element has the following *attributes*:

- 407 • **name : NCName (1..1)** – name of the method being decorated. The *@name* attribute of a
408 *<function/>* child element of a *<implementation.cpp/>* MUST be unique amongst the *<function/>*
409 elements of that *<implementation.cpp/>*. [CPP20010]
- 410 • **allowsPassByReference : boolean (0..1)** – indicates the method allows pass by reference data
411 exchange semantics.

412 2.5 Instantiation

413 A C++ implementation class MUST be default constructable by the SCA runtime to instantiate the
414 component. [CPP20011]

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.

421 The following snippet shows the `ComponentContext` C++ class with its `getService()` member
422 function.

```
423 namespace oasis {  
424     namespace sca {  
425  
426         class ComponentContext {  
427             public:  
428                 static ComponentContextPtr getCurrent();  
429                 virtual void * getService(  
430                     const std::string& referenceName) const = 0;  
431                 ...  
432             }  
433         }  
434     }  
435 }
```

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

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

```
445  
446 #include "ComponentContext.h"  
447 #include "CustomerService.h"  
448  
449 using namespace oasis::sca;  
450  
451 void clientFunction()  
452 {  
453  
454     unsigned long customerNumber = 1234;  
455  
456     ComponentContextPtr context = ComponentContext::getCurrent();  
457  
458     CustomerService* service =  
459         (CustomerService* )context->getService("customerService");  
460  
461     short rating = service->getCreditRating(customerNumber);  
462  
463 }
```

464 **3.2 Accessing Services from non-SCA component implementations**

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

467

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

470 **3.3 Calling Service Operations**

471 The previous sections show the various options for getting access to a service. Once you have access to
472 the service, calling an operation of the service is like calling a member function of a C++ class.

473

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

477

4 Asynchronous Programming

478 Asynchronous programming of a service is where a client invokes a service and carries on executing
479 without waiting for the service to execute. Typically, the invoked service executes at some later time.
480 Output from the invoked service, if any, is fed back to the client through a separate mechanism, since no
481 output is available at the point where the service is invoked. This is in contrast to the call-and-return style
482 of synchronous programming, where the invoked service executes and returns any output to the client
483 before the client continues. The SCA asynchronous programming model consists of support for non-
484 blocking operation calls and callbacks. Each of these topics is discussed in the following sections.

4.1 Non-blocking Calls

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

489

490 Any member function that returns `void`, has only by-value parameters and has no declared exceptions
491 can be marked with the `@oneWay="true"` attribute in the interface definition of the service. A member
492 function marked as `oneWay="true"` is considered non-blocking and the SCA runtime MAY use a binding
493 that buffers the requests to the member function and sends them at some time after they are made.

494 **[CPP40001]**

495

496 The following snippet shows the component type for a service with the `reportEvent()` member
497 function declared as a one-way operation:

498

```
499 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">  
500   <service name="LoanService">  
501     <interface.cpp header="LoanService.h">  
502       <function name="reportEvent" oneWay="true" />  
503     </interface.cpp>  
504   </service>  
505 </componentType>
```

506

507 SCA does not currently define a mechanism for making non-blocking calls to methods that return values
508 or are declared to throw exceptions. It is considered to be a best practice that service designers define
509 one-way member function as often as possible, in order to give the greatest degree of binding flexibility to
510 deployers.

4.2 Callbacks

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

514

515 A callback interface is declared by the `@callbackHeader` and `@callbackClass` attributes in the interface
516 definition of the service. The following snippet shows the component type for a service `MyService` with the
517 interface defined in `MyService.h` and the interface for callbacks defined in `MyServiceCallback.h`,

518

```
519 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" >  
520   <service name="MyService">  
521     <interface.cpp header="MyService.h"  
522       callbackHeader="MyServiceCallback.h"/>  
523   </service>
```

524 </componentType>

525 4.2.1 Using Callbacks

526 Bidirectional interfaces and callbacks are used when a simple request/response pattern isn't sufficient to
527 capture the business semantics of a service interaction. Callbacks are well suited for cases when a
528 service request can result in multiple responses or new requests from the service back to the client, or
529 where the service might respond to the client some time after the original request has completed.

530

531 The following example shows a scenario in which bidirectional interfaces and callbacks could be used. A
532 client requests a quotation from a supplier. To process the enquiry and return the quotation, some
533 suppliers might need additional information from the client. The client does not know which additional
534 items of information will be needed by different suppliers. This interaction can be modeled as a
535 bidirectional interface with callback requests to obtain the additional information.

536

```
537 class Quotation {  
538 public:  
539     virtual double requestQuotation(std::string productCode,  
540                                     unsigned int quantity) = 0;  
541 };  
542  
543 class QuotationCallback {  
544 public:  
545     virtual std::string getState() = 0;  
546     virtual std::string getZipCode() = 0;  
547     virtual std::string getCreditRating() = 0;  
548 };
```

549

550 In this example, the `requestQuotation` operation requests a quotation to supply a given quantity of a
551 specified product. The `QuotationCallBack` interface provides a number of operations that the supplier can
552 use to obtain additional information about the client making the request. For example, some suppliers
553 might quote different prices based on the state or the zip code to which the order will be shipped, and
554 some suppliers might quote a lower price if the ordering company has a good credit rating. Other
555 suppliers might quote a standard price without requesting any additional information from the client.

556

557 The following code snippet illustrates a possible implementation of the example service.

558

```
559 #include "QuotationImpl.h"  
560 #include "QuotationCallback.h"  
561 #include "oasis/sca/ComponentContext.h"  
562 using namespace oasis::sca;  
563  
564 double QuotationImpl::requestQuotation(std::string productCode,  
565                                         unsigned int quantity) {  
566     double price = getPrice(productCode, quantity);  
567     double discount = 0;  
568  
569     ComponentContextPtr context = ComponentContext::getCurrent();  
570     ServiceReferencePtr serviceRef = context->getSelfReference();  
571     QuotationCallback* callback =  
572         (QuotationCallback* ) serviceRef->getCallback();  
573     if (quantity > 1000 && callback->getState().compare("FL") == 0)  
574         discount = 0.05;  
575     if (quantity > 10000 && callback->getCreditRating().data() == 'A')  
576         discount += 0.05;  
577     return price * (1-discount);  
578 }
```

579

580 The code snippet below is taken from the client of this example service. The client's service
581 implementation class implements the methods of the QuotationCallback interface as well as those of its
582 own service interface ClientService.

583

```
584 #include "QuotationImpl.h"  
585 #include "QuotationCallback.h"  
586 #include "oasis/sca/ComponentContext.h"  
587 using namespace oasis::sca;  
588  
589 void ClientImpl:: aClientFunction() {  
590     ComponentContextPtr context = ComponentContext::getCurrent();  
591  
592     service = (QuotationService* )context->getService("quotationService");  
593  
594     service->requestQuotation("AB123", 2000);  
595 }  
596  
597 std::string QuotationCallbackImpl::getState() {  
598     return "TX";  
599 }  
600 std::string QuotationCallbackImpl::getZipCode() {  
601     return "78746";  
602 }  
603 std::string QuotationCallbackImpl::getCreditRating() {  
604     return "AA";  
605 }
```

606

607 In this example the callback is **stateless**, i.e., the callback requests do not need any information relating
608 to the original service request. For a callback that needs information relating to the original service
609 request (a **stateful** callback), this information can be passed to the client by the service provider as
610 parameters on the callback request..

611 4.2.2 Callback Instance Management

612 Instance management for callback requests received by the client of the bidirectional service is handled in
613 the same way as instance management for regular service requests. If the client implementation has
614 STATELESS scope, the callback is dispatched using a newly initialized instance. If the client
615 implementation has COMPOSITE scope, the callback is dispatched using the same shared instance that
616 is used to dispatch regular service requests.

617 As described **Error! Reference source not found.**, a stateful callback can obtain information relating to
618 the original service request from parameters on the callback request. Alternatively, a composite-scoped
619 client could store information relating to the original request as instance data and retrieve it when the
620 callback request is received. These approaches could be combined by using a key passed on the
621 callback request (e.g., an order ID) to retrieve information that was stored in a composite-scoped instance
622 by the client code that made the original request.

623 4.2.3 Implementing Multiple Bidirectional Interfaces

624 Since it is possible for a single class to implement multiple services, it is also possible for callbacks to be
625 defined for each of the services that it implements. To access the callbacks the
626 ServiceReference::getCallback(serviceName) member function must be used, passing in the
627 name of the service for which the callback is to be obtained.

628 5 Error Handling

629 Clients calling service operations will experience business exceptions, and SCA runtime exceptions.

630

631 Business exceptions are raised by the implementation of the called service operation. It is expected that
632 these will be caught by client invoking the operation on the service.

633

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

- 637 • `SCAException` – defines a root exception type from which all SCA defined exceptions derive.
- 638 – `SCANullPointerException` – signals that code attempted to dereference a null pointer from a
639 `RefCountingPointer` object.
- 640 – `ServiceRuntimeException` - signals problems in the management of the execution of SCA
641 components.
- 642 • `ServiceUnavailableException` – signals problems in the interaction with remote
643 services. This extends `ServiceRuntimeException`. These are exceptions that may be
644 transient, so retrying is appropriate. Any exception that is a `ServiceRuntimeException`
645 that is not a `ServiceUnavailableException` is unlikely to be resolved by retrying the
646 operation, since it most likely requires human intervention.
- 647 • `MultipleServicesException` – signals that a member function expecting identification of
648 a single service is called where there are multiple services defined. Thrown by
649 `ComponentContext::getService()`, `ComponentContext::getSelfReference()`
650 and `ComponentContext::getServiceReference()`.

651 6 C++ API

652 All the C++ interfaces are found in the namespace `oasis::sca`, which has been omitted from the
653 following descriptions for clarity.

654

655 An SCA runtime MUST implement Reference Counting Pointers, the ComponentContext, Service
656 Reference and SCAExceptions classes. [CPP60001]

657 6.1 Reference Counting Pointers

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

662

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

665

666 `RefCountingPointer` defines member functions with raw pointer like semantics. This includes
667 defining operators for dereferencing the pointer (`*`, `->`), as well as operators for determining the validity of
668 the pointer.

669

```
670 template <typename T>  
671 class RefCountingPointer {  
672 public:  
673     T& operator* () const;  
674     T* operator-> () const;  
675     operator void* () const;  
676     bool operator! () const;  
677 };
```

678

679 The *RefCountingPointer* class has the following member functions:

680 6.1.1 operator*

681 A C++ component implementation uses the `*` operator to dereferences the underlying pointer of a
682 reference counting pointer. This is equivalent to calling `*p` where `p` is the underlying pointer.

Precondition	C++ component instance is running and has a reference counting pointer
Return	A reference to the value of the pointer
Throws	<code>SCANullPointerException</code> if the pointer is NULL
Post Condition	No change

683 6.1.2 operator->

684 A C++ component implementation uses the `->` operator to invoke member functions on the underlying
685 pointer of a reference counting pointer. This is equivalent to invoking `p->func()` where `func()` is a
686 member function defined on the underlying pointer type.

Precondition	C++ component instance is running and has a reference counting pointer
Return	
Throws	SCANullPointerException if the pointer is NULL
Post Condition	The underlying member functions has been processed.

687 **6.1.3 operator void***

688 A C++ component implementation uses the `void*` operator to determine if the underlying pointer of a
689 reference counting pointer is set, i.e. `if (p) { /* do something */ }`.

Precondition	C++ component instance is running and has a reference counting pointer
Return	Zero if the underlying pointer is null, otherwise a non-zero value
Post Condition	No change

690 **6.1.4 operator!**

691 A C++ component implementation uses the `!` operator to determine if the underlying pointer of a
692 reference counting pointer is not set, i.e. `if (!p) { /* do something */ }`.

Precondition	C++ component instance is running and has a reference counting pointer
Return	A non-zero value if the underlying pointer is null, otherwise zero
Post Condition	No change

693 **6.2 Component Context**

694 The following shows the `ComponentContext` interface.

695

```

696 class ComponentContext {
697 public:
698     static ComponentContextPtr getCurrent();
699
700     virtual std::string getURI() const = 0;
701
702     virtual void * getService(const std::string& referenceName) const = 0;
703     virtual std::list<void*> getServices(
704         const std::string& referenceName) const = 0;
705
706     virtual ServiceReferencePtr getServiceReference(
707         const std::string& referenceName) const = 0;
708     virtual std::list<ServiceReferencePtr> getServiceReferences(
709         const std::string& referenceName) const = 0;
710
711
712     virtual DataObjectPtr getProperties() const = 0;
713     virtual DataFactoryPtr getDataFactory() const = 0;
714
715     virtual ServiceReferencePtr getSelfReference() const = 0;
716     virtual ServiceReferencePtr getSelfReference(
717         const std::string& serviceName) const = 0;
718 };

```

719

720 The **ComponentContext** C++ interface has the following member functions:

721 6.2.1 **getCurrent**

722 A C++ component implementation uses `ComponentContext::getCurrent()` to get a
723 `ComponentContext` object for itself.

Precondition	C++ component instance is running
Input Parameter	
Return	<code>ComponentContext</code> for the current component
Post Condition	The component instance has a valid context object to use for subsequent runtime calls.

724 6.2.2 **getURI**

725 A C++ component implementation uses `getURI()` to get an absolute URI for itself.

Precondition	C++ component instance is running and has a <code>ComponentContext</code>
Input Parameter	
Return	Absolute URI for the current component
Post Condition	No change

726 6.2.3 **getService**

727 A C++ component implementation uses `getService()` to get an object implementing the interface
728 defined for a Reference.

Precondition	C++ component instance is running and has a <code>ComponentContext</code>
Input Parameter	<code>referenceName</code> Name of the Reference to get an interface object for
Return	Pointer to an object implementing the interface of the Reference. This will be NULL if <code>referenceName</code> is not defined for the component.
Throws	<code>MultipleServicesException</code> if the reference resolves to more than one service
Post Condition	An interface object for the Reference is constructed. This interface object is independent of any <code>ServiceReference</code> that may be obtained for the Reference.

729 6.2.4 **getServices**

730 A C++ component implementation uses `getServices()` to get a list of object implementing the interface
731 defined for a Reference.

Precondition	C++ component instance is running and has a <code>ComponentContext</code>
Input Parameter	<code>referenceName</code> Name of the Reference to get an interface object for
Return	List of pointers to objects implementing the interface of the Reference. This list will be empty if <code>referenceName</code> is not defined for the component. Operations must be invoked on each object in the list.
Post Condition	Interface objects for the Reference are constructed. These interface objects are independent of any <code>ServiceReferences</code> that may be obtained for the Reference.

732 **6.2.5 getServiceReference**

733 A C++ component implementation uses `getServiceReference()` to get a `ServiceReference` for a
734 Reference.

Precondition	C++ component instance is running and has a <code>ComponentContext</code>	
Input Parameter	<code>referenceName</code>	Name of the Reference to get a <code>ServiceReference</code> for
Return	<code>ServiceReference</code> for the Reference. This will be NULL if <code>referenceName</code> is not defined for the component.	
Throws	<code>MultipleServicesException</code> if the reference resolves to more than one service	
Post Condition	A <code>ServiceReference</code> for the Reference is constructed.	

735 **6.2.6 getServiceReferences**

736 A C++ component implementation uses `getServiceReferences()` to get a list of
737 `ServiceReference` for a Reference.

Precondition	C++ component instance is running and has a <code>ComponentContext</code>	
Input Parameter	<code>referenceName</code>	Name of the Reference to get a list of <code>ServiceReferences</code> for
Return	List of <code>ServiceReferences</code> for the Reference. This will be empty if <code>referenceName</code> is not defined for the component.	
Post Condition	<code>ServiceReferences</code> for the Reference are constructed.	

738 **6.2.7 getProperties**

739 A C++ component implementation uses `getProperties()` to get its configured property values.

Precondition	C++ component instance is running and has a <code>ComponentContext</code>	
Input Parameter		
Return	An SDO [SDO21] from which all the properties defined in the <code>componentType</code> file can be retrieved.	
Post Condition	An SDO with the property values for the component instance is constructed.	

740 **6.2.8 getDataFactory**

741 A C++ component implementation uses `getDataFactory()` to get its an SDO `DataFactory` which
742 can be used to create `DataObjects` for complex data types used by this component.

Precondition	C++ component instance is running and has a <code>ComponentContext</code>	
Input Parameter		
Return	An SDO <code>DataFactory</code> which has definitions for all complex data types used by a component.	
Post Condition	An SDO <code>DataFactory</code> is constructed	

743 **6.2.9 getSelfReference**

744 A C++ component implementation uses `getSelfReference()` to get a `ServiceReference` for use
745 with some callback APIs.

746

747 There are two variations of this API.

Precondition	C++ component instance is running and has a <code>ComponentContext</code>	
Input Parameter		
Return	A <code>ServiceReference</code> for the service provided by this component.	
Throws	<code>MultipleServicesException</code> if the component implements more than one <code>Service</code>	
Post Condition	A <code>ServiceReference</code> object is constructed	

748 and

Precondition	C++ component instance is running and has a <code>ComponentContext</code>	
Input Parameter	<code>serviceName</code>	Name of the <code>Service</code> to get a <code>ServiceReference</code> for
Return	A <code>ServiceReference</code> for the service provided by this component.	
Post Condition	A <code>ServiceReference</code> object is constructed	

749 **6.3 ServiceReference**

750 The following shows the `ServiceReference` interface.

751

```

752 class ServiceReference {
753 public:
754     virtual void* getService() const = 0;
755
756     virtual void* getCallback() const = 0;
757 };

```

758

759 The ***ServiceReference*** interface has the following member functions (the detailed description of the
760 usage of these member functions is described in the section Asynchronous Programming):

761 **6.3.1 getService**

762 A C++ component implementation uses `getService()` to get an object implementing the interface
763 defined for a `ServiceReference`.

Precondition	C++ component instance is running and has a <code>ServiceReference</code>	
Input Parameter		
Return	Pointer to an object implementing the interface of the <code>ServiceReference</code> .	
Post Condition	An interface object for the <code>ServiceReference</code> is constructed.	

764 6.3.2 getCallback

765 A C++ component implementation uses `getCallback()` to get an object implementing the callback
766 interface defined for a `ServiceReference`.

Precondition	C++ component instance is running and has a <code>ServiceReference</code>
Input Parameter	
Return	Pointer to an object implementing the callback interface of the <code>ServiceReference</code> .
Throws	<code>NoRegisteredCallbackException</code> if no callback interface is defined.
Post Condition	An interface object for the callback interface of the <code>ServiceReference</code> is constructed.

767 6.4 SCAException

768 The following shows the `SCAException` interface.

769

```
770 class SCAException : public std::exception {  
771     public:  
772         const char* getEClassName() const;  
773         const char* getMessageText() const;  
774         const char* getFileName() const;  
775         unsigned long getLineNumber() const;  
776         const char* getFunctionName() const;  
777     };
```

778

779 The **SCAException** C++ interface has the following member functions (the details concerning this class
780 and its derived types are described in the section **Error! Reference source not found.**):

781 6.4.1 getEClassName

782 A C++ component implementation uses `getEClassName()` to get the name of the exception type.

Precondition	C++ component instance is running and has caught an SCA Exception
Input Parameter	
Return	The type of the exception as a string. e.g. "ServiceUnavailableException"
Post Condition	No change

783 6.4.2 getMessageText

784 A C++ component implementation uses `getMessageText()` to get any message included with the
785 exception.

Precondition	C++ component instance is running and has caught an SCA Exception
Input Parameter	
Return	The message which the SCA runtime attached to the exception
Post Condition	No change

786 6.4.3 getFileName

787 A C++ component implementation uses `getFileName()` to get the filename containing the function
788 where the exception occurred.

Precondition	C++ component instance is running and has caught an SCA Exception
Input Parameter	
Return	The filename within which the exception occurred – Will be an empty string if the filename is not known
Post Condition	No change

789 6.4.4 getLineNumber

790 A C++ component implementation uses `getLineNumber()` to get the line number in the source file
791 where the exception occurred.

Precondition	C++ component instance is running and has caught an SCA Exception
Input Parameter	
Return	The line number at which the exception occurred – Will 0 if the line number is not known
Post Condition	No change

792 6.4.5 getFunctionName

793 A C++ component implementation uses `getFunctionName()` to get the function name where the
794 exception occurred.

Precondition	C++ component instance is running and has caught an SCA Exception
Input Parameter	
Return	The function name in which the exception occurred – Will be an empty string if the function name is not known
Post Condition	No change

795 6.5 SCANullPointerException

796 The following shows the `SCANullPointerException` interface.

797

```
798 class SCANullPointerException : public SCAException {  
799     };  
800
```

801 6.6 ServiceRuntimeException

802 The following shows the `ServiceRuntimeException` interface.

803

```
804 class ServiceRuntimeException : public SCAException {  
805     };  
806
```

806 **6.7 ServiceUnavailableException**

807 The following shows the `ServiceUnavailableException` interface.

808

```
809 class ServiceUnavailablException : public ServiceRuntimeException {  
810     };
```

811 **6.8 MultipleServicesException**

812 The following shows the `MultipleServicesException` interface.

813

```
814 class MultipleServicesException : public ServiceRuntimeException {  
815     };
```

816

7 C++ Contributions

817 Contributions are defined in the Assembly specification [ASSEMBLY]. C++ contributions are typically, but
818 not necessarily contained in .zip files. In addition to SCDL and potentially WSDL artifacts, C++
819 contributions include binary executable files, componentType files and potentially C++ interface headers.
820 No additional discussion is needed for header files, but here are some additional considerations for
821 executable and componentType files discussed in the following sections.

7.1 Executable files

823 Executable files containing the C++ implementations for a contribution can be contained in the
824 contribution, contained in another contribution or external to any contribution. In some cases, it could be
825 desirable to have contributions share an executable. In other cases, an implementation deployment
826 policy might dictate that executables are placed in specific directories in a file system.

7.1.1 Executable in contribution

828 When the executable file containing a C++ implementation is in the same contribution, the *@path*
829 attribute of the *implementation.cpp* element is used to specify the location of the executable. The specific
830 location of an executable within a contribution is not defined by this specification.

831

832 The following shows a contribution containing a DLL.

833

```
834 META-INF/  
835   sca-contribution.xml  
836 bin/  
837   autoinsurance.dll  
838 AutoInsurance/  
839   AutoInsurance.composite  
840   AutoInsuranceService/  
841     AutoInsurance.h  
842     AutoInsuranceImpl.componentType  
843 include/  
844   Customers.h  
845   Underwriting.h  
846   RateUtils.h
```

847

848 The SCDL for the AutoInsuranceService component is:

849

```
850 <component name="AutoInsuranceService">  
851   <implementation.cpp library="autoinsurance" path="bin/"  
852     class="AutoInsuranceImpl" />  
853 </component>
```

7.1.2 Executable shared with other contribution(s) (Export)

855 If a contribution contains an executable that also implements C++ components found in other
856 contributions, the contribution has to export the executable. An executable in a contribution is made
857 visible to other contributions by adding an *export.cpp* element to the contribution definition as shown in
858 the following snippet.

859

```
860 <contribution>  
861   <deployable composite="myNS:RateUtilities"
```

```
862     <export.cpp name="contribNS:rates" >
863 </contribution>
```

864

865 It is also possible to export only a subtree of a contribution. If a contribution contains the following:

866

```
867 META-INF/
868     sca-contribution.xml
869 bin/
870     rates.dll
871 RateUtilities/
872     RateUtilities.composite
873     RateUtilitiesService/
874         RateUtils.h
875         RateUtilsImpl.componentType
```

876

877 An export of the form:

878

```
879 <contribution>
880     <deployable composite="myNS:RateUtilities"
881     <export.cpp name="contribNS:ratesbin" path="bin/" >
882 </contribution>
```

883

884 only makes the contents of the bin directory visible to other contributions. By placing all of the executable
885 files of a contribution in a single directory and exporting only that directory, the amount of information
886 contribution that uses the exported executable files is limited. This is considered a best practice.

887 7.1.3 Executable outside of contribution (Import)

888 When the executable that implements a C++ component is located outside of a contribution, the
889 contribution MUST import the executable. If the executable is located in another contribution, the
890 **import.cpp** element of the contribution definition uses a *@location* attribute that identifies the name of the
891 export as defined in the contribution that defined the export as shown in the following snippet.

892

```
893 <contribution>
894     <deployable composite="myNS:Underwriting"
895     <import.cpp name="rates" location="contribNS:rates">
896 </contribution>
```

897

898 The SCDL for the UnderwritingService component is:

899

```
900 <component name="UnderwritingService">
901     <implementation.cpp library="rates" path="rates:bin/"
902         class="UnderwritingImpl" />
903 </component>
```

904

905 If the executable is located in the file system, the *@location* attribute identifies the location in the files
906 system used as the root of the import as shown in this snippet.

907

```
908 <contribution>
909     <deployable composite="myNS:CustomerUtilities"
910     <import.cpp name="usr-bin" location="/usr/bin/" >
911 </contribution>
```

912 7.2 componentType files

913 As stated in section 2.5, each component implemented in C++ has a corresponding componentType file.
914 This componentType file is, by default, located in the root directory of the composite containing the
915 component or a subdirectory of the composite root with the name *<implementation
916 class>.componentType*, as shown in the following example.

```
917  
918 META-INF/  
919   sca-contribution.xml  
920 bin/  
921   autoinsurance.dll  
922 AutoInsurance/  
923   AutoInsurance.composite  
924   AutoInsuranceService/  
925     AutoInsurance.h  
926     AutoInsuranceImpl.componentType
```

927
928 The SCDL for the AutoInsuranceService component is:

```
929  
930 <component name="AutoInsuranceService">  
931   <implementation.cpp library="autoinsurance" path="bin/"  
932     class="AutoInsuranceImpl" />  
933 </component>
```

934
935 Since there is a one-to-one correspondence between implementations and componentTypes, when an
936 implementation is shared between contributions, it is desirable to also share the componentType file.
937 ComponentType files can be exported and imported in the same manner as executable files. The
938 location of a *.componentType* file can be specified using the *@componentType* attribute of the
939 *implementation.cpp* element.

```
940  
941 <component name="UnderwritingService">  
942   <implementation.cpp library="rates" path="rates:bin/"  
943     class="UnderwritingImpl" componentType="rates:types/UnderwritingImpl"  
944   />  
945 </component>
```

946 7.3 C++ Contribution Extensions

947 7.3.1 Export.cpp

948 The following snippet shows the schema for the C++ export element used to make an executable or
949 componentType file visible outside of a contribution.

```
950  
951 <?xml version="1.0" encoding="ASCII"?>  
952 <!-- export.cpp schema snippet -->  
953 <export.cpp xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
954   name="QName" path="string"? >
```

955
956 The *export.cpp* element has the following *attributes*:

- 957 • **name : QName (1..1)** – name of the export. The *@name* attribute of a *<export.cpp/>* element MUST
958 be unique amongst the *<export.cpp/>* elements in a domain. [CPP70001]
- 959 • **path : string (0..1)** – path of the exported executable relative to the root of the contribution. If not
960 present, the entire contribution is exported.

961 7.3.2 Import.cpp

962 The following snippet shows the schema for the C++ import element used to reference an executable or
963 componentType file that is outside of a contribution.

964

```
965 <?xml version="1.0" encoding="ASCII"?>  
966 <!-- import.cpp schema snippet -->  
967 <import.cpp xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
968     name="QName" location="string" >
```

969

970 The *import.cpp* element has the following *attributes*:

- 971 • **name : QName (1..1)** – name of the import. The *@name* attribute of a *<import.cpp/>* child element of
972 a *<contribution/>* MUST be unique amongst the *<import.cpp/>* elements in of that contribution.
973 [\[CPP70002\]](#)
- 974 • **location : string (1..1)** – either the QName of a export or a file system location. If the value does not
975 match an export name it is taken as an absolute file system path.

976 8 Types Supported in Service Interfaces

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

979 8.1 Local service

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

- 981 • Any of the C++ primitive types (for example, `int`, `short`, `char`). In this case the types will be passed
982 by value as is normal for C++.
- 983 • Pointers to any of the C++ primitive types (for example, `int *`, `short *`, `char *`).
- 984 • The `const` keyword can be used for any pointer to a C++ primitive type (for example `const char`
985 `*`). If this is used on a parameter then the destination may not change the value.
- 986 • C++ `class`. The class will be passed by value as is normal for C++.
- 987 • Pointer to a C++ `class`. A pointer will be passed to the destination which can then modify the original
988 contents.
- 989 • `DataObjectPtr`. An SDO pointer. This will be passed by reference.
- 990 • References to C++ classes (passed by reference)

991 8.2 Remotable service

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

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

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

- 1000 • Pointers.
- 1001 • References.

1002 9 Restrictions on C++ header files

1003 A C++ header file that is used to describe an interface has some restrictions.

1004 A C++ header file used to define an interface MUST:

- 1005 • Declare at least one class with:
 - 1006 – At least one public member function.
 - 1007 – All public member functions must be pure virtual (virtual with no implementation) [CPP90001]

1008

1009 A C++ header file used to define an interface MUST NOT use the following constructs:

- 1010 • Macros
- 1011 • Inline member functions
- 1012 • Friend classes [CPP90002]

10 WSDL to C++ and C++ to WSDL Mapping

1013

1014 The SCA Client and Implementation Model for C++ applies the WSDL to Java and Java to WSDL
1015 mapping rules (augmented for C++) as defined by the JAX-WS specification [JAXWS21] for generating
1016 remotable C++ interfaces from WSDL portTypes and vice versa. Use of the JAX-WS specification as a
1017 guideline for WSDL to C++ and C++ to WSDL mappings should not imply that any support for the Java
1018 language is required by this specification.

1019

1020 For the purposes of the Java-to-WSDL mapping algorithm, the interface is treated as if it had a
1021 @WebService annotation on the class, even if it doesn't. For the WSDL-to-Java mapping, the generated
1022 @WebService annotation implies that the interface is @Remotable.

1023

1024 For the mapping from C++ types to XML schema types SCA supports the SDO 2.1 [SDO21] mapping. A
1025 detailed mapping of C++ to WSDL types and WSDL to C++ types is covered in section SDO Data
1026 Binding.

1027

1028 The following limitations apply:

- 1029 • JAX-WS style external binding files are not supported. (See JAX-WS Sec. 2)
- 1030 • MIME binding is not supported. (See JAX-WS Sec. 2.1.1)
- 1031 • Holder classes are not supported. (See JAX-WS Sec. 2.3.3)
- 1032 • Asynchronous mapping is not supported. (See JAX-WS Sec. 2.3.4)
- 1033 • Generation of Service classes from WSDL is not supported. (See JAX-WS Sec. 2.7)
- 1034 • Generation of WSDL from Service implementation classes is not supported (See JAX-WS Sec. 3.3)
- 1035 • Templates are not supported when converting from C++ to WSDL (See JAX-WS Sec. 3.9)

1036

1037 The following general rules apply to the application of JAX-WS to C++.

- 1038 • References to Java should be considered references to C++.
- 1039 • References to Java classes should be considered references to C++ classes.
- 1040 • References to Java methods should be considered references to C++ member functions.
- 1041 • References to Java interfaces should be considered references to C++ classes which only define
1042 pure virtual member functions.
- 1043 • For the purposes of the C++-to-WSDL mapping algorithm, a C++ class with only pure-virtual functions
1044 and no state is treated as if it had a @WebService annotation on the class. All default values are
1045 assumed for the @WebService annotation.

1046

1047 Major divergences from JAX-WS:

- 1048 • Algorithms for converting WSDL namespaces to C++ namespaces (and vice-versa).
- 1049 • Mapping of WSDL faults to C++ exceptions and vice-versa.
- 1050 • Managing of data bindings.

1051

1052 10.1 Augmentations for WSDL to C++ Mapping

1053 10.1.1 Mapping WSDL targetNamespace to a C++ namespace

1054 Since C++ does not define a standard convention for the use of namespaces, the SCA specification does
1055 not define an implicit mapping of WSDL targetNamespaces to C++ namespaces. A WSDLfile might
1056 define a namespace using the <sca:namespace> WSDL extension, otherwise all C++ classes MUST be
1057 placed in a default namespace as determined by the implementation. Implementations SHOULD provide
1058 a mechanism for overriding the default namespace. [CPP100001]

1059 10.1.2 Mapping WSDL Faults to C++ Exceptions

1060 WSDL operations that specify one or more <wsdl:fault> elements will produce a C++ member function
1061 that is annotated with an @WebThrows annotation listing a C++ exception class associated with each
1062 <wsdl:fault>.

1063

1064 The C++ exception class associated with a fault will be generated based on the message that is
1065 associated with the <wsdl:fault> element, and in particular with the global element that the
1066 wsdl:fault/wsdl:message/@part indicates.

1067

```
1068 FaultException(const char* message, const FaultInfo& faultInfo);  
1069 FaultInfo getFaultInfo() const;
```

1070

1071 Where *FaultException* is the name of the generated exception class, and where *FaultInfo* is the name of
1072 the C++ type representing the fault's global element type.

1073 10.1.2.1 Multiple Fault References

1074 If multiple operations within the same portType indicate that they throw faults that reference the same
1075 global element, an SCA implementation MUST generate a single C++ exception class with each C++
1076 member function referencing this class in its @WebThrows annotation. [CPP100002]

1077 10.1.3 Mapping of in, out, in/out parts to C++ member function 1078 parameters

1079 C++ diverges from the JAX-WS specification in it's handling of some parameter types, especially around
1080 how passing of out and in/out parameters are handled in the context of C++'s various pass-by styles.
1081 The following outlines an updated mapping for use with C++.

- 1082 • For unwrapped messages, an SCA implementation MUST map:
 - 1083 – **in** - the message part to a member function parameter, passed by const-reference.
 - 1084 – **out** - the message part to a member function parameter, passed by reference, or to the member
1085 function return type, returned by-value.
 - 1086 – **in/out** - the message part to a member function parameter, passed by reference. [CPP100003]
 - 1087
- 1088 • For wrapped messages, an SCA implementation MUST map:
 - 1089 – **in** - the wrapper child to a member function parameter, passed by const-reference.
 - 1090 – **out** - the wrapper child to a member function parameter, passed by reference, or to the member
1091 function return type, returned by-value.
 - 1092 – **in/out** - the wrapper child to a member function parameter, passed by reference. [CPP100004]
 - 1093

1094 10.2 Augmentations for C++ to WSDL Mapping

1095 10.2.1 Mapping C++ namespaces to WSDL namespaces

1096 Since C++ does not define a standard convention for the use of namespaces, the SCA specification does
1097 not define an implicit mapping of C++ namespaces to WSDL namespace URIs. The default
1098 targetNamespace is defined by the implementation. An SCA implementation SHOULD provide a
1099 mechanism for overriding the default targetNamespace. [CPP100005]

1100 10.2.2 Parameter and return type classification

1101 The classification of parameters and return types in C++ are determined based on how the value is
1102 passed into the function.

1103
1104 An SCA implementation MUST map a method's return type as an **out** parameter, a parameter passed by-
1105 reference or by-pointer as an **in/out** parameter, and all other parameters, including those passed by-
1106 const-reference as **in** parameters. [CPP100006]

1107
1108 An application can customize parameter classification using the @WebParam annotation.

1109 10.2.3 C++ to WSDL Type Conversion

1110 C++ types are mapped to WSDL and schema types based on the mapping described in Section Simple
1111 Content Binding.

1112 10.2.4 Service-specific Exceptions

1113 C++ classes that define a web service interface can indicate which faults they may throw using the
1114 @WebThrows annotation. @WebThrows lists the names of each C++ class that might be thrown as a
1115 fault from a particular member function. An SCA implementation must ensure each class that is
1116 referenced from an @WebThrows annotation MUST itself have a @WebFault annotation that associates
1117 the fault with a particular global element that will be associated with the fault message. [CPP100007]

1118

1119 10.3 SDO Data Binding

1120 10.3.1 Simple Content Binding

1121 The translation of XSD simple content types to C++ types follows the convention defined in the SDO
1122 specification. The following table summarizes that mapping as it applies to SCA services.

1123

1124 The following mapping is derived from the mappings for SDO types to XSD schema types [SDO 11.1],
1125 XSD schema types to SDO types [SDO 10.3.3], and SDO types to C++ types [SDO 9.1].

1126

XSD Schema Type →	SDO Type →	C++ Type	→ XSD Schema Type
anySimpleType	Object	UNDEFINED	
anyType	DataObject	commonj::sdo::DataObject	anyType
anyURI	URI	std::string	string
base64Binary	Bytes	char*	hexBinary

Boolean	Boolean	bool	Boolean
Byte	Byte	int8_t	Byte
Date	YearMonthDay	std::string	string
dateTime	DateTime	std::string	string
Decimal	Decimal	<i>UNDEFINED</i>	
Double	Double	long double	double
Duration	Duration	std::string	string
ENTITIES	Strings	<i>UNDEFINED</i>	
ENTITY	String	std::string	string
Float	Float	float	float
gDay	Day	std::string	string
gMonth	Month	std::string	string
gMonthDay	MonthDay	std::string	string
gYear	Year	std::string	string
gYearMonth	YearMonth	std::string	string
hexBinary	Bytes	char*	hexBinary
ID	String	std::string	string
IDREF	String	std::string	string
IDREFS	Strings	<i>UNDEFINED</i>	
Int	Int	int32_t	int
Integer	Integer	<i>UNDEFINED</i>	
language	String	std::string	string
Long	Long	int64_t	long
Name	String	std::string	string
NCName	String	std::string	string
negativeInteger	Integer	<i>UNDEFINED</i>	
NMTOKEN	String	std::string	string
NMTOKENS	Strings	<i>UNDEFINED</i>	
nonNegativeInteger	Integer	<i>UNDEFINED</i>	
nonPositiveInteger	Integer	<i>UNDEFINED</i>	
normalizedString	String	std::string	string
NOTATION	String	std::string	string
positiveInteger	Integer	<i>UNDEFINED</i>	

QName	URI	std::string	string
short	Short	int16_t	short
string	String	std::string	string
time	Time	std::string	string
Token	String	std::string	string
unsignedByte	Short	int16_t	short
unsignedInt	Long	int64_t	long
unsignedLong	Integer	<i>UNDEFINED</i>	
unsignedShort	Int	int32_t	int

1127

1128 **10.3.2 Complex Content Binding**

1129 Any XSD complex types are mapped to an instance of an SDO DataObject.

1130 11 Conformance

1131 This section specifies the conformance targets of this specification and the requirements that apply to
1132 each of them.

1133 11.1 Conformance Targets

1134 The conformance targets of this specification are:

- 1135 • **SCA implementations**, which provide a **runtime** for SCA components and potentially **tools** for
1136 authoring SCA artifacts, component descriptions and/or runtime operations.
- 1137 • **SCDL documents**, which describe SCA artifacts, and specific **elements** within these documents.
- 1138 • **C++ component implementations**, which execute under the control of an SCA runtime.
- 1139 • **C++ header files**, which are used to define SCA service interfaces.
- 1140 • **WSDL files**, which are used to define SCA service interfaces.

1141 11.2 Conformance Claims

1142 A claim of conformance with this specification **MUST** meet the following requirements:

- 1143 • It **MUST** state which conformance targets it implements.

1144 11.3 SCA Implementations

1145 An implementation conforms to this specification if it meets the following conditions:

- 1146 1. It **MUST** conform to the SCA Assembly Model Specification [**ASSEMBLY**] and the SCA Policy
1147 Framework [**POLICY**].
- 1148 2. It **MUST** implement the SCA C++ API defined in section C++ API.
- 1149 3. It **MUST** implement the mapping between C++ and WSDL 1.1 [**WSDL11**] as described in WSDL to
1150 C++ and C++ to WSDL Mapping.
- 1151 4. It **MUST** support C++ contributions as defined in C++ Contributions
- 1152 5. It **MAY** support source file annotations as defined in C++ SCA Annotations, C++ SCA Policy
1153 Annotations and C++ WSDL Mapping Annotations.

1154 11.4 SCDL Documents

1155 A SCDL file conforms to this specification if it meets the following conditions:

- 1156 1. It **MUST** conform to the SCA Assembly Model Specification [**ASSEMBLY**] and, if appropriate, the
1157 SCA Policy Framework [**POLICY**].
- 1158 2. It conforms to the requirements in section Component Type and Component or section C++
1159 Contributions according to the document type.

1160 11.5 C++ Component Implementations

1161 A C++ component implementation conforms to this specification if it meets the following conditions:

- 1162 1. It conforms to the requirements for a C++ component implementation specified in section Basic
1163 Component Implementation Model.

1164 11.6 C++ Header Files

1165 A C++ header files conforms to this specification if it meets the following conditions:

- 1166 1. It conforms to the requirements and restrictions for a C++ header file specified in section Restrictions
1167 on C++ header files.
- 1168 2. If it contains annotations, the formats and restrictions in section C++ SCA Annotations, C++ SCA
1169 Policy Annotations and C++ WSDL Mapping Annotations are followed.
- 1170

1171 **11.7 WSDL Files**

1172 A WSDL conforms to this specification if it meets the following conditions:

- 1173 3. It is a valid WSDL 1.1 [**WSDL11**] document.
- 1174 4. If it contains C++ WSDL extensions, the restrictions in section WSDL C++ Mapping Extensions are
1175 followed.

1176 **11.8 Extensions**

1177 [What extension points do we want to identify?].

1178

A C++ SCA Annotations

1179 To allow developers to define SCA related information directly in source files, without having to separately
1180 author SCDL files, a set of annotations are defined. An SCA implementation MAY support source file
1181 annotations. If annotations are supported by an implementation, the annotations defined here MUST be
1182 supported and MUST be mapped to SCDL as described. The SCA runtime MUST only process the
1183 SCDL files and not the annotations. [CPPA0001]

1184

1185 The annotations are defined as C++ comments in interface and implementation header files, for example:

1186

1187

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

1188

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

1189 In general an annotation immediately precedes the program element it applies to. If multiple annotations
1190 apply to a program element, all of the annotations SHOULD be in the same comment block. [CPPA0002]

1191

- Class

1192

The annotation immediately precedes the class.

1193

Example:

1194

1195

1196

1197

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

1198

- Member function

1199

The annotation immediately precedes the member function.

1200

Example:

1201

1202

1203

1204

1205

1206

1207

```
class LoanService
{
public:
// @OneWay
virtual void reportEvent(int eventId) = 0;
    ...
};
```

1208

- Data Member

1209

The annotation immediately precedes the data member.

1210

Example:

1211

1212

1213

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

1214

Annotations follow normal inheritance rules. An annotation on a base class or any element of a base
1215 class applies to any classes derived from the base class.

1216

A.2 Interface Header Annotations

1217

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

1218 A.2.1 @Interface

1219 Annotation used to indicate a class defines an interface when multiple classes exist in a header file. An
1220 SCA implementation MUST treat a class with an @WebService annotation specified. [CPA0003]

1221

1222

1223 **Corresponds to:** @class attribute of an *interface.cpp* element.

1224

1225 **Format:**

```
1226 // @Interface
```

1227

1228 **Applies to:** Class

1229

1230 **Example:**

1231 **Interface header:**

```
1232 // @Interface
1233 class LoanService {
1234     ...
1235 };
```

1236

1237 **Service definition:**

```
1238 <service name="LoanService">
1239     <interface.cpp header="LoanService.h" class="LoanService" />
1240 </service>
```

1241 A.2.2 @Remotable

1242 Annotation on service interface class to indicate that a service is remotable.

1243

1244 **Corresponds to:** @remotable="true" attribute of an *interface.cpp* element.

1245

1246 **Format:**

```
1247 // @Remotable
```

1248 The default is **false** (not remotable).

1249

1250 **Applies to:** Class

1251

1252 **Example:**

1253 **Interface header:**

```
1254 // @Remotable
1255 class LoanService {
1256     ...
1257 };
```

1258

1259 **Service definition:**

```
1260 <service name="LoanService">
1261     <interface.cpp header="LoanService.h" remotable="true" />
1262 </service>
```

1263 **A.2.3 @Callback**

1264 Annotation on a service interface class to specify the callback interface.

1265

1266 **Corresponds to:** `@callbackHeader` and `@callbackClass` attributes of an *interface.cpp* element.

1267

1268 **Format:**

```
1269 // @Callback(header="headerName", class="className")
```

1270 where

- 1271 • **headerName : (1..1)** – is the name of the header defining the callback service interface.
- 1272 • **className : (0..1)** – is the name of the class for the callback interface.

1273

1274 **Applies to:** Class

1275

1276 Example:

1277 Interface header:

```
1278 // @Callback(header="MyServiceCallback.h", class="MyServiceCallback")
1279 class MyService {
1280 public:
1281     virtual void someFunction( unsigned int arg ) = 0;
1282 };
```

1283

1284 Service definition:

```
1285 <service name="MyService">
1286     <interface.cpp header="MyService.h"
1287         callbackHeader="MyServiceCallback.h"
1288         callbackClass="MyServiceCallback" />
1289 </service>
```

1290 **A.2.4 @OneWay**

1291 Annotation on a service interface member function to indicate the member function is one way. The
1292 `@OneWay` annotation also affects the representation of a service in WSDL, see `@OneWay`.

1293

1294 **Corresponds to:** `@oneWay="true"` attribute of function element of an *interface.cpp* element.

1295

1296 **Format:**

```
1297 // @OneWay
```

1298 The default is **false** (not OneWay).

1299

1300 **Applies to:** Member function

1301

1302 Example:

1303 Interface header:

```
1304 class LoanService
1305 {
1306 public:
1307     // @OneWay
1308     virtual void reportEvent(int eventId) = 0;
```

1309 ...
1310 };

1311

1312 Service definition:

```
1313 <service name="LoanService">  
1314   <interface.cpp header="LoanService.h">  
1315     <function name="reportEvent" oneWay="true" />  
1316   </interface.cpp>  
1317 </service>
```

1318 A.3 Implementation Header Annotations

1319 This section lists the annotations that can be used in the header file that defines a service
1320 implementation.

1321 A.3.1 @ComponentType

1322 Annotation used to indicate which class implements a componentType when multiple classes exist in an
1323 implementation file.

1324

1325 **Corresponds to:** @class attribute of an *implementation.cpp* element.

1326

1327 **Format:**

```
1328 // @ComponentType
```

1329

1330 **Applies to:** Class

1331

1332 **Example:**

1333 Implementation header:

```
1334 // @ComponentType  
1335 class LoanServiceImpl : public LoanService {  
1336   ...  
1337 };
```

1338

1339 Component definition:

```
1340 <component name="LoanService">  
1341   <implementation.cpp library="loan" class="LoanServiceImpl"  
1342     class="LoanServiceImpl" />  
1343 </component>
```

1344 A.3.2 @Scope

1345 Annotation on a service implementation class to indicate the scope of the service.

1346

1347 **Corresponds to:** @scope attribute of an *implementation.cpp* element.

1348

1349 **Format:**

```
1350 // @Scope("value")
```

1351 where

- 1352 • **value** : [*stateless* | *composite*] (1..1) – specifies the scope of the implementation. The default value
1353 is *stateless*.

1354

1355 **Applies to:** Class

1356

1357 Example:

1358 Implementation header:

```
1359 // @Scope("composite")
1360 class LoanServiceImpl : public LoanService {
1361     ...
1362 };
```

1363

1364 Component definition:

```
1365 <component name="LoanService">
1366     <implementation.cpp library="loan" class="LoanServiceImpl"
1367         scope="composite" />
1368 </component>
```

1369 **A.3.3 @EagerInit**

1370 Annotation on a service implementation class to indicate the implantation is to be instantiated when its
1371 containing component is started.

1372

1373 **Corresponds to:** `@eagerInit="true"` attribute of an *implementation.cpp* element.

1374

1375 **Format:**

```
1376 // @EagerInit
```

1377 The default is **false** (the service should be initialized lazily).

1378

1379 **Applies to:** Class

1380

1381 Example:

1382 Implementation header:

```
1383 // @EagerInit
1384 class LoanServiceImpl : public LoanService {
1385     ...
1386 };
```

1387

1388 Component definition:

```
1389 <component name="LoanService">
1390     <implementation.cpp library="loan" class="LoanServiceImpl"
1391         eagerInit="true" />
1392 </component>
```

1393 **A.3.4 @AllowsPassByReference**

1394 Annotation on service implementation class or member function to indicate that a service or member
1395 function allows pass by reference semantics.

1396

1397 **Corresponds to:** `@allowsPassByReference="true"` attribute of an *implementation.cpp* element or a
1398 *function* child element of an *implementation.cpp* element.

1399

1400 **Format:**

```
1401 // @AllowsPassByReference
```

1402 The default is **false** (the service does not allow by reference parameters).

1403

1404 **Applies to:** Class or Member function

1405

1406 **Example:**

1407 Implementation header:

```
1408 // @AllowsPassByReference
1409 class LoanService {
1410     ...
1411 };
```

1412

1413 Component definition:

```
1414 <component name="LoanService">
1415     <implementation.cpp library="loan" class="LoanServiceImpl"
1416         allowsPassByReference="true" />
1417 </component>
```

1418 **A.3.5 @Property**

1419 Annotation on a service implementation class data member to define a property of the service.

1420

1421 **Corresponds to:** *property* element of a *componentType* element.

1422

1423 **Format:**

```
1424 // @Property(name="propertyName", type="typeQName"
1425 //             default="defaultValue", required="true")
```

1426 where

- 1427 • **name : NCName (0..1)** - specifies the name of the property. If name is not specified the property name is taken from the name of the following data member.
- 1428
- 1429 • **type : QName (0..1)** - specifies the type of the property. If not specified the type of the property is based on the C++ mapping of the type of the following data member to an xsd type as defined in SDO Data Binding. If the data member is an array, then the property is many-valued.
- 1430
- 1431
- 1432 • **required : boolean (0..1)** - specifies whether a value has to be set in the component definition for this property. Default is *false*
- 1433
- 1434 • **default : <type> (0..1)** - specifies a default value and is only needed if *required* is *false*,

1435

1436 **Applies to:** DataMember

1437

1438 **Example:**

1439 Implementation:

```
1440 // @Property(name="loanType", type="xsd:int")
1441 long loanType;
```

1442

1443 Component Type definition:

```
1444 <componentType ... >
1445     <service ... />
```

```
1446     <property name="loanType" type="xsd:int" />
1447 </componentType>
```

1448 **A.3.6 @Reference**

1449 Annotation on a service implementation class data member to define a reference of the service.

1450

1451 **Corresponds to:** *reference* element of a *componentType* element.

1452

1453 **Format:**

```
1454     // @Reference (name="referenceName", interfaceHeader="LoanService.h",
1455     //             interfaceClass="LoanService", required="true")
```

1456 where

- 1457 • **name : NCName (0..1)** - specifies the name of the reference. If name is not specified the reference
- 1458 name is taken from the name of the following data member.
- 1459 • **interfaceHeader : Name (1..1)** - specifies the C++ header defining the interface for the reference.
- 1460 • **interfaceClass : Name (0..1)** - specifies the C++ class defining the interface for the reference. If not
- 1461 specified the class is derived from the type of the annotated data member.
- 1462 • **required : boolean (0..1)** - specifies whether a value has to be set for this reference. Default is *true*.

1463

1464 If the annotated data member is a `std::list` then the implied component type has a reference with a

1465 multiplicity of either 0..n or 1..n depending on the value of the `@Reference` *required* attribute – 1..n

1466 applies if *required=true*. Otherwise a multiplicity of 0..1 or 1..1 is implied.

1467

1468 **Applies to:** Data Member

1469

1470 Example:

1471 Implementation:

```
1472     // @Reference (interfaceHeader="LoanService.h" required="true")
1473     LoanService* loanService;
1474
1475     // @Reference (interfaceHeader="LoanService.h" required="false")
1476     std::list<LoanService*> loanServices;
```

1477

1478 Component Type definition:

```
1479     <componentType ... >
1480     <service ... />
1481     <reference name="loanService" multiplicity="1..1">
1482         <interface.cpp header="LoanService.h" class="LoanService" />
1483     </reference>
1484     <reference name="loanServices" multiplicity="0..n">
1485         <interface.cpp header="LoanService.h" class="LoanService" />
1486     </reference>
1487 </componentType>
```

1488 **A.4 Base Annotation Grammar**

```
1489     <annotation> ::= // @<baseAnnotation>
1490
1491     <baseAnnotation> ::= <name> [(<params>)]
1492
1493     <params> ::= <paramNameValue>[, <paramNameValue>]* |
```

```
1494         <paramValue>[, <paramValue>]*
1495
1496 <paramNameValue> ::= <name>="<value>"
1497
1498 <paramValue> ::= "<value>"
1499
1500 <name> ::= NCName
1501
1502 <value> ::= string
```

- 1503 • Adjacent string constants are concatenated
- 1504 • NCName is as defined by XML schema **[XSD]**
- 1505 • Whitespace including newlines between tokens is ignored.
- 1506 • Annotations with parameters may span multiple lines within a comment, and are considered complete
- 1507 when the terminating “)” is reached.

1508

B C++ SCA Policy Annotations

1509 SCA provides facilities for the attachment of policy-related metadata to SCA assemblies, which influence
1510 how implementations, services and references behave at runtime. The policy facilities are described in
1511 **[POLICY]**. In particular, the facilities include Intents and Policy Sets, where intents express abstract,
1512 high-level policy requirements and policy sets express low-level detailed concrete policies.

1513

1514 Policy metadata can be added to SCA assemblies through the means of declarative statements placed
1515 into Composite documents and into Component Type documents. These annotations are completely
1516 independent of implementation code, allowing policy to be applied during the assembly and deployment
1517 phases of application development.

1518

1519 However, it can be useful and more natural to attach policy metadata directly to the code of
1520 implementations. This is particularly important where the policies concerned are relied on by the code
1521 itself. An example of this from the Security domain is where the implementation code expects to run
1522 under a specific security Role and where any service operations invoked on the implementation must be
1523 authorized to ensure that the client has the correct rights to use the operations concerned. By annotating
1524 the code with appropriate policy metadata, the developer can rest assured that this metadata is not lost or
1525 forgotten during the assembly and deployment phases.

1526

1527 The SCA C++ policy annotations provide the capability for the developer to attach policy information to
1528 C++ implementation code. The annotations concerned first provide general facilities for attaching SCA
1529 Intents and Policy Sets to C++ code. Secondly, there are further specific annotations that deal with
1530 particular policy intents for certain policy domains such as Security.

1531 B.1 General Intent Annotations

1532 SCA provides the annotation **@Requires** for the attachment of any intent to a C++ class, to a C++
1533 interface or to elements within classes and interfaces such as member functions and data members.

1534

1535 The **@Requires** annotation can attach one or multiple intents in a single statement.

1536

1537 Each intent is expressed as a string. Intents are XML QNames, which consist of a Namespace URI
1538 followed by the name of the Intent. The precise form used is as follows:

1539

```
1540 {" + Namespace URI + "} + intentname
```

1541

1542 Intents may be qualified, in which case the string consists of the base intent name, followed by a ".",
1543 followed by the name of the qualifier. There may also be multiple levels of qualification.

1544

1545 This representation is quite verbose, so we expect that reusable constants will be defined for the
1546 namespace part of this string, as well as for each intent that is used by C++ code. SCA defines constants
1547 for intents such as the following:

1548

```
1549 // @Define SCA_PREFIX "{http://docs.oasis-  
1550 open.org/ns/opencsa/sca/200712}"  
1551 // @Define CONFIDENTIALITY SCA_PREFIX ## "confidentiality"  
1552 // @Define CONFIDENTIALITY_MESSAGE CONFIDENTIALITY ## ".message"
```

1553
1554 Notice that, by convention, qualified intents include the qualifier as part of the name of the constant,
1555 separated by an underscore. These intent constants are defined in the file that defines an annotation for
1556 the intent (annotations for intents, and the formal definition of these constants, are covered in a following
1557 section).

1558
1559 Multiple intents (qualified or not) are expressed as separate strings within an array declaration.
1560

1561 **Corresponds to:** *@requires* attribute of a *service*, *reference*, *operation* or *property* element.

1562
1563 **Format:**

```
1564 // @Requires("qualifiedIntent" | {"qualifiedIntent" [, "qualifiedIntent"]})
```

1565 where

```
1566 qualifiedIntent ::= QName | QName.qualifier | QName.qualifier1.qualifier2
```

1567
1568 **Applies to:** Class, Member function, Data Member
1569

1570 Examples:

1571 Attaching the intents "confidentiality.message" and "integrity.message".

```
1572 // @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
```

1573
1574 A reference requiring support for confidentiality:

```
1575 class Foo {  
1576     ...  
1577     // @Requires(CONFIDENTIALITY)  
1578     // @Reference(interfaceHeader="SetBar.h")  
1579     void setBar(Bar* bar);  
1580     ...  
1581 }
```

1582
1583 Users may also choose to only use constants for the namespace part of the QName, so that they may
1584 add new intents without having to define new constants. In that case, this definition would instead look
1585 like this:

```
1586  
1587 class Foo {  
1588     ...  
1589     // @Requires(SCA_PREFIX "confidentiality ")  
1590     // @Reference(interfaceHeader="SetBar.h")  
1591     void setBar(Bar* bar);  
1592     ...  
1593 }
```

1594 B.2 Specific Intent Annotations

1595 In addition to the general intent annotation supplied by the *@Requires* annotation described above, there
1596 are C++ annotations that correspond to some specific policy intents.

1597
1598 The general form of these specific intent annotations is an annotation with a name derived from the name
1599 of the intent itself. If the intent is a qualified intent, qualifiers are supplied as an attribute to the annotation
1600 in the form of a string or an array of strings.

1601

1602 For example, the SCA confidentiality intent described in General Intent Annotations using the
1603 @Requires(CONFIDENTIALITY) intent can also be specified with the specific @Confidentiality intent
1604 annotation. The specific intent annotation for the "integrity" security intent is:

```
1605 // @Integrity
```

1606

1607 **Corresponds to:** @requires="<Intent>" attribute of a *service*, *reference*, *operation* or *property* element.

1608

1609 **Format:**

```
1610 // @<Intent>[(qualifiers)]
```

1611 where Intent is an NCName that denotes a particular type of intent.

```
1612 Intent ::= NCName  
1613 qualifiers ::= "qualifier " | {"qualifier" [, "qualifier"] }  
1614 qualifier ::= NCName | NCName/qualifier
```

1615

1616 **Applies to:** Class, Member function, Data Member – but see specific intents for restrictions

1617

1618 **Example:**

```
1619 // @Authentication( {"message", "transport"} )
```

1620 This annotation attaches the pair of qualified intents: *authentication.message* and *authentication.transport*
1621 (the sca: namespace is assumed in both of these cases – "http://docs.oasis-
1622 open.org/opencsa/ns/sca/200712").

1623 The Policy Framework **[POLICY]** defines a number of intents and qualifiers. The following sections
1624 define the annotations for those intents.

1625 B.2.1 Security Interaction

Intent	Annotation
authentication	@Authentication
confidentiality	@Confidentiality
integrity	@Integrity

1626

1627 These three intents can be qualified with

- 1628 • transport
- 1629 • message

1630 B.2.2 Security Implementation

Intent	Annotation
runAs	@RunAs(role"role")
Allow	@Allow(roles="<comma separated list of roles>")
permitAll	@PermitAll
denyAll	@DenyAll

1631

1632 In addition to allow roles to defined, an SCA runtime MAY use the following annotation
1633 @DeclareRoles(<comma separated list of roles>”)

1634 B.2.3 Reliable Messaging

Intent	Annotation
atLeastOnce	@AtLeastOnce
atMostOnce	@AtMostOnce
Ordered	@Ordered
exactlyOnce	@ExactlyOnce

1635

1636 B.2.4 Transactions

Intent	Annotation	Qualifiers
managedTransaction	@ManagedTransaction	Local global
transactedOneWay	@TransactedOneWay	
immediateOneWay	@ImmediateOneWay	
propagates Transaction	@PropagatesTransaction	
suspendsTransaction	@SuspendsTransaction	

1637

1638 B.2.5 Miscellaneous

Intent	Annotation	Qualifiers
SOAP	@SOAP	1_1 1_2
JMS	@JMS	

1639 B.3 Application of Intent Annotations

1640 Where multiple intent annotations (general or specific) are applied to the same C++ element, they are
1641 additive in effect. An example of multiple policy annotations being used together follows:

1642

```
1643 // @Authentication  
1644 // @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
```

1645

1646 In this case, the effective intents are *authentication*, *confidentiality.message* and *integrity.message*.

1647

1648 If an annotation is specified at both the class/interface level and the member function or data member
1649 level, then the member function or data member level annotation completely overrides the class level
1650 annotation of the same type.

1651
1652 The intent annotation can be applied either to classes or to class member functions when adding
1653 annotated policy on SCA services.

1654 B.4 Inheritance and Intent Annotations

1655 The following example shows the inheritance relations of intents on classes, operations, and super
1656 classes.

```
1657  
1658 // @Remotable  
1659 // @Integrity("transport")  
1660 // @Authentication  
1661 class HelloService {  
1662 public:  
1663 // @Integrity  
1664 // @Authentication("message")  
1665 wchar_t* hello(wchar_t* message) {...}  
1666  
1667 // @Integrity  
1668 // @Authentication("transport")  
1669 wchar_t* helloThere() {...}  
1670 }  
1671  
1672 // @Remotable  
1673 // @Confidentiality("message")  
1674 class HelloChildService : public HelloService {  
1675 public:  
1676 // @Confidentiality("transport")  
1677 wchar_t* hello(wchar_t* message) {...}  
1678 // @Authentication  
1679 wchar_t* helloWorld(){...}  
1680 }
```

1681 Example 1a. Usage example of annotated policy and inheritance.

- 1682
- 1683 • The effective intent annotation on the helloWorld member function is @Integrity("transport"),
1684 @Authentication, and @Confidentiality("message").
 - 1685 • The effective intent annotation on the hello member function of the HelloChildService is
1686 @Integrity("transport"), @Authentication, and @Confidentiality("transport"),
 - 1687 • The effective intent annotation on the helloThere member function of the HelloChildService is
1688 @Integrity and @Authentication("transport"), the same as in HelloService class.
 - 1689 • The effective intent annotation on the hello member function of the HelloService is @Integrity and
1690 @Authentication("message")

1691
1692 The listing below contains the equivalent declarative security interaction policy of the HelloService and
1693 HelloChildService implementation corresponding to the C++ classes shown in Example 1a.

```
1694  
1695 <?xml version="1.0" encoding="ASCII"?>  
1696  
1697 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"  
1698 name="HelloServiceComposite" >  
1699 <service name="HelloService " requires="integrity/transport  
1700 authentication">  
1701 ...  
1702 </service>  
1703 <service name="HelloChildService" requires="integrity/transport  
1704 authentication confidentiality/message">
```

```

1705 ...
1706 </service>
1707 ...
1708
1709 <component name="HelloServiceComponent">*
1710     <implementation.cpp library="HelloService.dll"
1711         class="HelloServiceImpl"/>
1712     <operation name="hello" requires="integrity
1713         authentication/message"/>
1714     <operation name="helloThere" requires="integrity
1715         authentication/transport"/>
1716 </component>
1717 <component name="HelloChildServiceComponent">*
1718     <implementation.cpp library="HelloChildService.dll"
1719         class="HelloChildServiceImpl" />
1720     <operation name="hello" requires="confidentiality/transport"/>
1721     <operation name="helloThere " requires="integrity/transport
1722         authentication"/>
1723     <operation name="helloWorld" requires="authentication"/>
1724 </component>
1725 ...
1726 ...
1727
1728 </composite>

```

1729 Example 1b. Declaratives intents equivalent to annotated intents in Example 1a.

1730 B.5 Relationship of Declarative and Annotated Intents

1731 Annotated intents on a C++ class cannot be overridden by declarative intents either in a composite
1732 document which uses the class as an implementation or by statements in a componentType document
1733 associated with the class. This rule follows the general rule for intents that they represent fundamental
1734 requirements of an implementation.

1735

1736 An unqualified version of an intent expressed through an annotation in the C function or function
1737 declaration may be qualified by a declarative intent in a using composite document.

1738 B.6 Policy Set Annotations

1739 The SCA Policy Framework uses Policy Sets to capture detailed low-level concrete policies (for example,
1740 a concrete policy is the specific encryption algorithm to use when encrypting messages when using a
1741 specific communication protocol to link a reference to a service).

1742 Policy Sets can be applied directly to C++ implementations using the **@PolicySets** annotation. The
1743 PolicySets annotation either takes the QName of a single policy set as a string or the name of two or
1744 more policy sets as an array of strings.

1746

1747 **Corresponds to:** *@policySets* attribute of a *service*, *reference*, *operation* or *property* element.

1748

1749 **Format:**

```

1750 // @PolicySets("<policy set QName>" |
1751     { "<policy set QName>" [, "<policy set QName>"] })

```

1752 As for intents, PolicySet names are QNames – in the form of "{Namespace-URI}localPart".

1753

1754 **Applies to:** Class, Member function, Data Member

1755

1756 **Example:**

```

1757 // @Reference(name="helloService", interfaceHeader="helloService.h",
1758 //           required="true")
1759 // @PolicySets({ MY_NS "WS_Encryption_Policy",
1760 //              MY_NS "WS_Authentication_Policy"})
1761 HelloService* helloService;
1762 ...

```

1763

1764 In this case, the Policy Sets `WS_Encryption_Policy` and `WS_Authentication_Policy` are applied, both
 1765 using the namespace defined for the constant `MY_NS`.

1766

1767 PolicySets must satisfy intents expressed for the implementation when both are present, according to the
 1768 rules defined in **[POLICY]**.

1769 B.7 Policy Annotation Grammar Additions

```

1770 <annotation> ::= // @<baseAnnotation> | @<requiresAnnotation> |
1771                @<intentAnnotation> | @<policySetAnnotation>
1772
1773 <requiresAnnotation> ::= Requires(<intents>)
1774
1775 <intents> ::= "<qualifiedIntent>" |
1776             {"<qualifiedIntent>"[, "<qualifiedIntent>"]*}
1777
1778 <qualifiedIntent> ::= <intentName> | <intentName>.<qualifier> |
1779                    <intentName>.<qualifier>.<qualifier>
1780
1781 <intentName> ::= {aAnyURI}NCName
1782
1783 <intentAnnotation> ::= <intent>[(<qualifiers>)]
1784
1785 <intent> ::= NCName [(param)]
1786
1787 <qualifiers> ::= "<qualifier>" | {"<qualifier>"[, "<qualifier>"]*}
1788
1789 <qualifier> ::= NCName | NCName/<qualifier>
1790
1791 <policySetAnnotation> ::= policySets(<policysets>)
1792
1793 <policysets> ::= "<policySetName>" | {"<policySetName>"[, "<policySetName>"]*}
1794
1795 <policySetName> ::= {aAnyURI}NCName

```

- 1796 • anyURI is as defined by XML schema **[XSD]**

1797 B.8 Annotation Constants

```

1798 <annotationConstant> ::= // @Define <identifier> <token string>
1799
1800 <identifier> ::= token
1801
1802 <token string> ::= "string" | "string"[ ## <token string>]

```

- 1803 • Constants are immediately expanded

1804

C C++ WSDL Mapping Annotations

1805 To allow developers to control the mapping of C++ to WSDL, a set of annotations are defined. An SCA
1806 implementation MAY support source file annotations for WSDL. If annotations are supported by an
1807 implementation, the annotations defined here MUST be supported and MUST be mapped to WSDL as
1808 described. [CPPC0001]

1809 C.1 Interface Header Annotations

1810 C.1.1 @WebService

1811 Annotation on a C++ class indicating that it represents a web service. An SCA implementation MUST
1812 treat any instance of a @Interface annotation and without an explicit @WebService annotation as if a
1813 @WebService annotation with no parameters was specified. An SCA implementation MUST treat any
1814 instance of a @Interface annotation and without an explicit @WebService annotation as if a
1815 @WebService annotation with no parameters was specified. An SCA implementation MUST treat any
1816 instance of a @Interface annotation and without an explicit @WebService annotation as if a
1817 @WebService annotation with no parameters was specified. [CPPC0002]

1818

1819 **Corresponds to:** javax.jws.WebService annotation in the JAX-WS specification (7.11.1)

1820

1821 Format:

```
1822 // @WebService (name="portTypeName", targetNamespace="namespaceURI",  
1823 //      serviceName="WSDLServiceName", portName="WSDLPortName")
```

1824 where:

- 1825 • **name : NCName (0..1)** – specifies the name of the web service portType. The default is the name of
1826 the C++ class the annotation is applied to.
- 1827 • **targetNamespace : anyURI (0..1)** – specifies the target namespace for the web service. The default
1828 namespace is determined by the implementation.
- 1829 • **serviceName : NCName (0..1)** – specifies the target name for the associated service. The default
1830 service name is the name of the C++ class suffixed with “Service”. The name of the associated
1831 binding is also determined by the serviceName. In the case of a SOAP binding, the binding name is
1832 the name of the service suffixed with “SoapBinding”.
- 1833 • **portName : NCName (0..1)** – specifies the name that should be used for the associated WSDL port
1834 for the service. If a @WebService does not have a **portName** element, an SCA implementation
1835 MUST use the value associated with the **name** element, suffixed with “Port”. [CPPC0003]

1836

1837 **Applies to:** Class

1838

1839 Example:

1840 Input C++ source file:

```
1841 // @WebService (name="StockQuote", targetNamespace="http://www.example.org/",  
1842 //      serviceName="StockQuoteService")  
1843 class StockQuoteService {  
1844     };
```

1845

1846 Generated WSDL file:

```
1847 <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
```

```

1848     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
1849     xmlns:tns="http://www.example.org/"
1850     xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
1851     targetNamespace="http://www.example.org/"
1852
1853     <portType name="StockQuote">
1854         <cpp:bindings>
1855             <cpp:class name="StockQuoteService"/>
1856         </cpp:bindings>
1857     </portType>
1858
1859     <binding name="StockQuoteServiceSoapBinding">
1860         <soap:binding style="document"
1861             transport="http://schemas.xmlsoap.org/soap/http"/>
1862     </binding>
1863
1864     <service name="StockQuoteService">
1865         <port name="StockQuotePort" binding="tns:StockQuoteServiceSoapBinding">
1866             <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
1867         </port>
1868     </service>
1869 </definitions>

```

1870 C.1.2 @WebFunction

1871 Annotation on a C++ member function indicating that it represents a web service operation.

1872

1873 **Corresponds to:** javax.jws.WebMethod annotation in the JAX-WS specification (7.11.2)

1874

1875 **Format:**

```

1876     // @WebFunction(operationName="operation", action="SOAPAction",
1877     //     exclude="false")

```

1878 where:

- 1879 • **operationName : NCName (0..1)** – specifies the name of the WSDL operation to associate with this
- 1880 function. The default is the name of the C++ member function the annotation is applied to.
- 1881 • **action : string (0..1)** – specifies the value associated with the soap:operation/@soapAction attribute
- 1882 in the resulting code. The default value is an empty string.
- 1883 • **exclude : boolean (0..1)** – specifies whether this member function should be included in the web
- 1884 service interface. The default value is *“false”*.

1885

1886 **Applies to:** Member function.

1887

1888 **Example:**

1889 **Input C++ source file:**

```

1890     // @WebService(name="StockQuote", targetNamespace="http://www.example.org/"
1891     //     serviceName="StockQuoteService")
1892     class StockQuoteService {
1893
1894         // @WebFunction(operationName="GetLastTradePrice",
1895         //     action="urn:GetLastTradePrice")
1896         float getLastTradePrice(const std::string& tickerSymbol);
1897
1898         // @WebFunction(exclude=true)
1899         void setLastTradePrice(const std::string& tickerSymbol, float value);
1900     };

```

1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963

Generated WSDL file:

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.example.org/"
  xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
  targetNamespace="http://www.example.org/"

  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://www.example.org/"
    attributeFormDefault="unqualified"
    elementFormDefault="unqualified"
    targetNamespace="http://www.example.org/"
    <xs:element name="GetLastTradePrice" type="tns:GetLastTradePrice"/>
    <xs:element name="GetLastTradePriceResponse"
      type="tns:GetLastTradePriceResponse"/>
    <xs:complexType name="GetLastTradePrice">
      <xs:sequence>
        <xs:element name="tickerSymbol" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="GetLastTradePriceResponse">
      <xs:sequence>
        <xs:element name="return" type="xs:float"/>
      </xs:sequence>
    </xs:complexType>
  </xs:schema>

  < message name="GetLastTradePrice">
    <part name="parameters" element="tns:GetLastTradePrice">
    </part>
  </message>

  < message name="GetLastTradePriceResponse">
    <part name="parameters" element="tns:GetLastTradePriceResponse">
    </part>
  </ message>

  <portType name="StockQuote">
    <cpp:bindings>
      <cpp:class name="StockQuoteService"/>
    </cpp:bindings>
    <operation name="GetLastTradePrice">
      <cpp:bindings>
        <cpp:memberFunction name="getLastTradePrice"/>
      </cpp:bindings>
      <input name="GetLastTradePrice" message="tns:GetLastTradePrice">
      </input>
      <output name="GetLastTradePriceResponse"
        message="tns:GetLastTradePriceResponse">
      </output>
    </operation>
  </portType>

  <binding name="StockQuoteServiceSoapBinding">
    <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="GetLastTradePrice">
      <soap:operation soapAction="urn:GetLastTradePrice" style="document"/>
      <wsdl:input name="GetLastTradePrice">
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="GetLastTradePriceResponse">
```

```

1964         <soap:body use="literal"/>
1965     </wsdl:output>
1966 </wsdl:operation>
1967 </binding>
1968
1969     <service name="StockQuoteService">
1970         <port name="StockQuotePort" binding="tns:StockQuoteServiceSoapBinding">
1971             <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
1972         </port>
1973     </service>
1974 </definitions>

```

1975 C.1.3 @OneWay

1976 Annotation on a C++ member function indicating that it represents a one-way request. The @OneWay
 1977 annotation also affects the service interface, see @OneWay.

1978

1979 **Corresponds to:** javax.jws.OneWay annotation in the JAX-WS specification (7.11.3)

1980

1981 **Format:**

```
1982 // @OneWay
```

1983

1984 **Applies to:** Member function.

1985

1986 **Example:**

1987 **Input C++ source file:**

```

1988 // @WebService(name="StockQuote", targetNamespace="http://www.example.org/"
1989 //     serviceName="StockQuoteService")
1990 class StockQuoteService {
1991
1992     // @WebFunction(operationName="GetLastTradePrice",
1993     //     action="urn:GetLastTradePrice")
1994     // @OneWay
1995     float getLastTradePrice(const std::string& tickerSymbol);
1996 };

```

1997

1998 **Generated WSDL file:**

```

1999 <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
2000     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
2001     xmlns:tns="http://www.example.org/"
2002     xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
2003     targetNamespace="http://www.example.org/"
2004
2005     <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
2006         xmlns:tns="http://www.example.org/"
2007         attributeFormDefault="unqualified"
2008         elementFormDefault="unqualified"
2009         targetNamespace="http://www.example.org/">
2010         <xs:element name="GetLastTradePrice" type="tns:GetLastTradePrice"/>
2011         <xs:complexType name="GetLastTradePrice">
2012             <xs:sequence>
2013                 <xs:element name="tickerSymbol" type="xs:string"/>
2014             </xs:sequence>
2015         </xs:complexType>
2016     </xs:schema>
2017
2018     < message name="GetLastTradePrice">

```

```

2019     <part name="parameters" element="tns:GetLastTradePrice">
2020     </part>
2021 </message>
2022
2023 <portType name="StockQuote">
2024     <cpp:bindings>
2025         <cpp:class name="StockQuoteService"/>
2026     </cpp:bindings>
2027     <operation name="GetLastTradePrice">
2028         <cpp:bindings>
2029             <cpp:memberFunction name="getLastTradePrice"/>
2030         </cpp:bindings>
2031         <input name="GetLastTradePrice" message="tns:GetLastTradePrice">
2032             </input>
2033         </operation>
2034     </portType>
2035
2036 <binding name="StockQuoteServiceSoapBinding">
2037     <soap:binding style="document"
2038         transport="http://schemas.xmlsoap.org/soap/http"/>
2039     <wsdl:operation name="GetLastTradePrice">
2040         <soap:operation soapAction="urn:GetLastTradePrice" style="document"/>
2041         <wsdl:input name="GetLastTradePrice">
2042             <soap:body use="literal"/>
2043         </wsdl:input>
2044     </wsdl:operation>
2045 </binding>
2046
2047 <service name="StockQuoteService">
2048     <port name="StockQuotePort" binding="tns:StockQuoteServiceSoapBinding">
2049         <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
2050     </port>
2051 </service>
2052 </definitions>

```

2053 C.1.4 @WebParam

2054 Annotation on a C++ member function parameter indicating its mapping to the associated input and
2055 output WSDL messages.

2056

2057 **Corresponds to:** javax.jws.WebParam annotation in the JAX-WS specification (7.11.4)

2058

2059 **Format:**

```

2060 // @WebParam(paramName=<="parameter", name="WSDLElement",
2061 //     targetNamespace="namespaceURI", mode="IN"|"OUT"|"INOUT",
2062 //     header="false", partName="WSDLPart", type="xsdType")

```

2063 where:

- 2064 • **paramName : NCName (1..1)** – specifies the name of the parameter that this annotation applies to.
2065 Only named parameters MAY be referenced by an @WebParam annotation. [CPPC0004]
- 2066 • **name : NCName (0..1)** – specifies the name of the associated WSDL part or element. The default
2067 value is the name of the parameter. If an @WebParam annotation is not present, and the parameter
2068 is unnamed, then a name of “argN”, where N is an incrementing value from 1 indicating the position of
2069 he parameter in the argument list, will be used.
- 2070 • **targetNamespace : string (0..1)** – specifies the target namespace for the part. The default
2071 namespace is the namespace of the associated @WebService. The targetNamespace attribute is
2072 ignored unless the binding style is document, and the binding parameterStyle is bare. See
2073 @SOAPBinding.

- 2074 • **mode : token (0..1)** – specifies whether the parameter is associated with the input message, output
2075 message, or both. The default value is determined by the passing mechanism for the parameter, see
2076 Parameter and return type classification.
- 2077 • **header : boolean (0..1)** – specifies whether this parameter is associated with a SOAP header
2078 element. The default value is “false”.
- 2079 • **partName : NCName (0..1)** – specifies the name of the WSDL part associated with this item. The
2080 default value is the value of name.
- 2081 • **type : NCName (0..1)** – specifies the XML Schema type of the WSDL part or element associated with
2082 this parameter. The value of the type property of a @WebParam annotation MUST be one of the
2083 simpleTypes defined in namespace http://www.w3.org/2001/XMLSchema. [CPPC0005] The default
2084 type is determined by the mapping defined in Simple Content Binding.

2085

2086 **Applies to:** Member function parameter.

2087

2088 **Example:**

2089 **Input C++ source file:**

```
2090 // @WebService(name="StockQuote", targetNamespace="http://www.example.org/"
2091 //     serviceName="StockQuoteService")
2092 class StockQuoteService {
2093
2094     // @WebFunction(operationName="GetLastTradePrice",
2095     //     action="urn:GetLastTradePrice")
2096     // @WebParam(paramName="tickerSymbol", name="symbol")
2097     float getLastTradePrice(const std::string& tickerSymbol);
2098 };
```

2099

2100 **Generated WSDL file:**

```
2101 <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
2102     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
2103     xmlns:tns="http://www.example.org/"
2104     xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
2105     targetNamespace="http://www.example.org/">
2106
2107     <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
2108         xmlns:tns="http://www.example.org/"
2109         attributeFormDefault="unqualified"
2110         elementFormDefault="unqualified"
2111         targetNamespace="http://www.example.org/">
2112         <xs:element name="GetLastTradePrice" type="tns:GetLastTradePrice"/>
2113         <xs:element name="GetLastTradePriceResponse"
2114             type="tns:GetLastTradePriceResponse"/>
2115         <xs:complexType name="GetLastTradePrice">
2116             <xs:sequence>
2117                 <xs:element name="symbol" type="xs:string"/>
2118             </xs:sequence>
2119         </xs:complexType>
2120         <xs:complexType name="GetLastTradePriceResponse">
2121             <xs:sequence>
2122                 <xs:element name="return" type="xs:float"/>
2123             </xs:sequence>
2124         </xs:complexType>
2125     </xs:schema>
2126
2127     <message name="GetLastTradePrice">
2128         <part name="parameters" element="tns:GetLastTradePrice">
2129             </part>
2130     </message>
```

```

2131
2132 < message name="GetLastTradePriceResponse">
2133   <part name="parameters" element="tns:GetLastTradePriceResponse">
2134     </part>
2135 </ message>
2136
2137 <portType name="StockQuote">
2138   <cpp:bindings>
2139     <cpp:class name="StockQuoteService"/>
2140   </cpp:bindings>
2141   <operation name="GetLastTradePrice">
2142     <cpp:bindings>
2143       <cpp:memberFunction name="getLastTradePrice"/>
2144       <cpp:parameter name="tickerSymbol"
2145         part="tns:GetLastTradePrice/parameter"
2146         childElementName="symbol"/>
2147     </cpp:bindings>
2148     <input name="GetLastTradePrice" message="tns:GetLastTradePrice">
2149       </input>
2150     <output name="GetLastTradePriceResponse"
2151       message="tns:GetLastTradePriceResponse">
2152       </output>
2153     </operation>
2154 </portType>
2155
2156 <binding name="StockQuoteServiceSoapBinding">
2157   <soap:binding style="document"
2158     transport="http://schemas.xmlsoap.org/soap/http"/>
2159   <wsdl:operation name="GetLastTradePrice">
2160     <soap:operation soapAction="urn:GetLastTradePrice" style="document"/>
2161     <wsdl:input name="GetLastTradePrice">
2162       <soap:body use="literal"/>
2163     </wsdl:input>
2164     <wsdl:output name="GetLastTradePriceResponse">
2165       <soap:body use="literal"/>
2166     </wsdl:output>
2167   </wsdl:operation>
2168 </binding>
2169
2170 <service name="StockQuoteService">
2171   <port name="StockQuotePort" binding="tns:StockQuoteServiceSoapBinding">
2172     <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
2173   </port>
2174 </service>
2175 </definitions>

```

2176 C.1.5 @WebResult

2177 Annotation on a C++ member function parameter indicating it's mapping to the associated output WSDL
 2178 messages.

2179

2180 **Corresponds to:** javax.jws.WebResult annotation in the JAX-WS specification (7.11.5)

2181

2182 Format:

```

2183 // @WebResult(name=<="WSDLElement", targetNamespace="namespaceURI",
2184 //   header="false", partName="WSDLPart", type="xsdType")

```

2185 where:

- 2186 • **name : NCName (0..1)** – specifies the name of the associated WSDL part or element. The default
 2187 value is "return".

- 2188 • **targetNamespace : string (0..1)** – specifies the target namespace for the part. The default
2189 namespace is the namespace of the associated @WebService. The targetNamespace attribute is
2190 ignored unless the binding style is document, and the binding parameterStyle is bare. See
2191 @SOAPBinding.
- 2192 • **header : boolean (0..1)** – specifies whether the result is associated with a SOAP header element.
2193 The default value is "false".
- 2194 • **partName : NCName (0..1)** – specifies the name of the WSDL part associated with this item. The
2195 default value is the value of name.
- 2196 • **type : NCName (0..1)** – specifies the XML Schema type of the WSDL part or element associated with
2197 this parameter. The value of the type property of a @WebResult annotation MUST be one of the
2198 simpleTypes defined in namespace http://www.w3.org/2001/XMLSchema. [CPPC0006] The default
2199 type is determined by the mapping defined in Simple Content Binding.

2200

2201 **Applies to:** Member function return value.

2202

2203 **Example:**

2204 **Input C++ source file:**

```
2205 // @WebService(name="StockQuote", targetNamespace="http://www.example.org/"
2206 //     serviceName="StockQuoteService")
2207 class StockQuoteService {
2208
2209     // @WebFunction(operationName="GetLastTradePrice",
2210     //     action="urn:GetLastTradePrice")
2211     // @WebResult(name="price")
2212     float getLastTradePrice(const std::string& tickerSymbol);
2213 };
```

2214

2215 **Generated WSDL file:**

```
2216 <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
2217     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
2218     xmlns:tns="http://www.example.org/"
2219     xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
2220     targetNamespace="http://www.example.org/">
2221
2222     <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
2223         xmlns:tns="http://www.example.org/"
2224         attributeFormDefault="unqualified"
2225         elementFormDefault="unqualified"
2226         targetNamespace="http://www.example.org/">
2227         <xs:element name="GetLastTradePrice" type="tns:GetLastTradePrice"/>
2228         <xs:element name="GetLastTradePriceResponse"
2229             type="tns:GetLastTradePriceResponse"/>
2230         <xs:complexType name="GetLastTradePrice">
2231             <xs:sequence>
2232                 <xs:element name="tickerSymbol" type="xs:string"/>
2233             </xs:sequence>
2234         </xs:complexType>
2235         <xs:complexType name="GetLastTradePriceResponse">
2236             <xs:sequence>
2237                 <xs:element name="price" type="xs:float"/>
2238             </xs:sequence>
2239         </xs:complexType>
2240     </xs:schema>
2241
2242     <message name="GetLastTradePrice">
2243         <part name="parameters" element="tns:GetLastTradePrice">
2244     </part>
```

```

2245 </message>
2246
2247 < message name="GetLastTradePriceResponse">
2248   <part name="parameters" element="tns:GetLastTradePriceResponse">
2249   </part>
2250 </ message>
2251
2252 <portType name="StockQuote">
2253   <cpp:bindings>
2254     <cpp:class name="StockQuoteService"/>
2255   </cpp:bindings>
2256   <operation name="GetLastTradePrice">
2257     <cpp:bindings>
2258       <cpp:memberFunction name="getLastTradePrice"/>
2259     </cpp:bindings>
2260     <input name="GetLastTradePrice" message="tns:GetLastTradePrice">
2261     </input>
2262     <output name="GetLastTradePriceResponse"
2263       message="tns:GetLastTradePriceResponse">
2264     </output>
2265   </operation>
2266 </portType>
2267
2268 <binding name="StockQuoteServiceSoapBinding">
2269   <soap:binding style="document"
2270     transport="http://schemas.xmlsoap.org/soap/http"/>
2271   <wsdl:operation name="GetLastTradePrice">
2272     <soap:operation soapAction="urn:GetLastTradePrice" style="document"/>
2273     <wsdl:input name="GetLastTradePrice">
2274       <soap:body use="literal"/>
2275     </wsdl:input>
2276     <wsdl:output name="GetLastTradePriceResponse">
2277       <soap:body use="literal"/>
2278     </wsdl:output>
2279   </wsdl:operation>
2280 </binding>
2281
2282 <service name="StockQuoteService">
2283   <port name="StockQuotePort" binding="tns:StockQuoteServiceSoapBinding">
2284     <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
2285   </port>
2286 </service>
2287 </definitions>

```

2288 C.1.6 @SOAPBinding

2289 Annotation on a C++ member function indicating that it represents a web service operation.

2290

2291 **Corresponds to:** javax.jws.SOAPBinding annotation in the JAX-WS specification (7.11.6)

2292

2293 **Format:**

```

2294 // @SOAPBinding(style="DOCUMENT"|"RPC", use="LITERAL"|"ENCODED",
2295 //   parameterStyle="BARE"|"WRAPPED")

```

2296 where:

- 2297 • **style : token (0..1)** – specifies the WSDL binding style. The default value is “DOCUMENT”.
- 2298 • **use : token (0..1)** – specifies the WSDL binding use. The default value is “LITERAL”.
- 2299 • **parameterStyle : token (0..1)** – specifies the WSDL parameter style. The default value is
2300 “WRAPPED”.

2301

2302 **Applies to:** Class, Member function.

2303

2304 **Example:**

2305 **Input C++ source file:**

```
2306 // @WebService(name="StockQuote", targetNamespace="http://www.example.org/"
2307 //     serviceName="StockQuoteService")
2308 // @SOAPBinding(style="RPC")
2309 class StockQuoteService {
2310 };
```

2311

2312 **Generated WSDL file:**

```
2313 <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
2314     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
2315     xmlns:tns="http://www.example.org/"
2316     xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
2317     targetNamespace="http://www.example.org/">
2318
2319     <portType name="StockQuote">
2320         <cpp:bindings>
2321             <cpp:class name="StockQuoteService"/>
2322         </cpp:bindings>
2323     </portType>
2324
2325     <binding name="StockQuoteServiceSoapBinding">
2326         <soap:binding style="document"
2327             transport="http://schemas.xmlsoap.org/soap/http"/>
2328     </binding>
2329
2330     <service name="StockQuoteService">
2331         <port name="StockQuotePort" binding="tns:StockQuoteServiceSoapBinding">
2332             <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
2333         </port>
2334     </service>
2335 </definitions>
```

2336

2337 **C.1.7 @WebFault**

2338 Annotation on a C++ exception class indicating that it may be thrown as a fault by a web service function.

2339 A C++ class with a `@WebFault` annotation **MUST** provide a constructor that takes two parameters, a `std::string` and a type representing the fault information. Additionally, the class **MUST** provide a const member function "getFaultInfo" that takes no parameters, and returns the same type as defined in the constructor. [\[CPPC0007\]](#)

2343

2344 **Corresponds to:** `javax.xml.ws.WebFault` annotation in the JAX-WS specification (7.2)

2345

2346 **Format:**

```
2347 // @WebFault(name="WSDLElement", targetNamespace="namespaceURI")
```

2348 where:

- 2349 • **name : NCName (1..1)** – specifies local name of the global element mapped to this fault.
- 2350 • **targetNamespace : string (0..1)** – specifies the namespace of the global element mapped to this fault. The default namespace is determined by the implementation.

2352

2353 **Applies to:** Class.

2354

2355 Example:

2356 Input C++ source file:

```
2357 // @WebFault (name="UnknownSymbolFault",
2358 //     targetNamespace="http://www.example.org/")
2359 class UnknownSymbol {
2360     UnknownSymbol(const char* message,
2361         const std::string& faultInfo);
2362
2363     std::string getFaultInfo() const;
2364 };
2365
2366 // @WebService (name="StockQuote", targetNamespace="http://www.example.org/"
2367 //     serviceName="StockQuoteService")
2368 class StockQuoteService {
2369
2370     // @WebFunction (operationName="GetLastTradePrice",
2371     //     action="urn:GetLastTradePrice")
2372     // @WebThrows (faults="UnknownSymbol")
2373     float getLastTradePrice(const std::string& tickerSymbol);
2374 };
```

2375

2376 Generated WSDL file:

```
2377 <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
2378     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
2379     xmlns:tns="http://www.example.org/"
2380     xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
2381     targetNamespace="http://www.example.org/">
2382
2383     <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
2384         xmlns:tns="http://www.example.org/"
2385         attributeFormDefault="unqualified"
2386         elementFormDefault="unqualified"
2387         targetNamespace="http://www.example.org/">
2388         <xs:element name="GetLastTradePrice" type="tns:GetLastTradePrice"/>
2389         <xs:element name="GetLastTradePriceResponse"
2390             type="tns:GetLastTradePriceResponse"/>
2391         <xs:complexType name="GetLastTradePrice">
2392             <xs:sequence>
2393                 <xs:element name="tickerSymbol" type="xs:string"/>
2394             </xs:sequence>
2395         </xs:complexType>
2396         <xs:complexType name="GetLastTradePriceResponse">
2397             <xs:sequence>
2398                 <xs:element name="return" type="xs:float"/>
2399             </xs:sequence>
2400         </xs:complexType>
2401         <xs:element name="UnknownSymbolFault" type="xs:string"/>
2402     </xs:schema>
2403
2404     <message name="GetLastTradePrice">
2405         <part name="parameters" element="tns:GetLastTradePrice">
2406             </part>
2407     </message>
2408
2409     <message name="GetLastTradePriceResponse">
2410         <part name="parameters" element="tns:GetLastTradePriceResponse">
2411             </part>
2412     </message>
2413
2414     <message name="UnknownSymbol">
```

```

2415     <part name="parameters" element="tns:UnknownSymbolFault">
2416     </part>
2417 </message>
2418
2419 <portType name="StockQuote">
2420   <cpp:bindings>
2421     <cpp:class name="StockQuoteService"/>
2422   </cpp:bindings>
2423   <operation name="GetLastTradePrice">
2424     <cpp:bindings>
2425       <cpp:memberFunction name="getLastTradePrice"/>
2426     </cpp:bindings>
2427     <input name="GetLastTradePrice" message="tns:GetLastTradePrice">
2428     </input>
2429     <output name="GetLastTradePriceResponse"
2430       message="tns:GetLastTradePriceResponse">
2431     </output>
2432     <fault name="UnknownSymbol" message="tns:UnknownSymbol">
2433     </fault>
2434   </operation>
2435 </portType>
2436
2437 <binding name="StockQuoteServiceSoapBinding">
2438   <soap:binding style="document"
2439     transport="http://schemas.xmlsoap.org/soap/http"/>
2440   <wsdl:operation name="GetLastTradePrice">
2441     <soap:operation soapAction="urn:GetLastTradePrice" style="document"/>
2442     <wsdl:input name="GetLastTradePrice">
2443       <soap:body use="literal"/>
2444     </wsdl:input>
2445     <wsdl:output name="GetLastTradePriceResponse">
2446       <soap:body use="literal"/>
2447     </wsdl:output>
2448     <wsdl:fault>
2449       <soap:fault name="UnknownSymbol" use="literal"/>
2450     </wsdl:fault>
2451   </wsdl:operation>
2452 </binding>
2453
2454 <service name="StockQuoteService">
2455   <port name="StockQuotePort" binding="tns:StockQuoteServiceSoapBinding">
2456     <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
2457   </port>
2458 </service>
2459 </definitions>

```

2460

2461 **C.1.8 @WebThrows**

2462 Annotation on a C++ class indicating which faults may be thrown by this class.

2463

2464 **Corresponds to:** No equivalent in JAX-WS.

2465

2466 **Format:**

```
2467 // @WebThrows (faults="faultMsg1" [, "faultMsgn"] *)
```

2468 where:

- 2469 • **faults : NMTOKEN (1..n)** – specifies the names of all faults that may be thrown by this member
2470 function. The name of the fault is the name of its associated C++ class name. A C++ class that is
2471 listed in a @WebThrows annotation MUST itself have a @WebFault annotation. **[CPPC0008]**

2472

2473 **Applies to:** Member function.

2474

2475 Example:

2476 See @WebFault.

2477

D WSDL C++ Mapping Extensions

2478 The following WSDL extensions are used to augment the conversion process from WSDL to C++. All of
2479 these extensions are defined in the namespace `http://docs.oasis-open.org/ns/opencsa/sca-c-`
2480 `cpp/cpp/200901`. For brevity, all definitions of these extensions will be fully qualified, and all references to
2481 the “cpp” prefix are associated with the namespace above. An SCA implementation MAY support these
2482 WSDL extensions. If these extensions are supported by an implementation, all the extensions defined
2483 here MUST be supported and MUST be mapped to C++ as described. [CPPD0001]
2484

2485 D.1 <cpp:bindings>

2486 <cpp:bindings> is a container type which may be used as a WSDL extension. All other SCA wsdl
2487 extensions will be specified as children of a <cpp:bindings> element. A <cpp:bindings> element may be
2488 used as an extension to any WSDL type that accepts extensions.

2489 D.2 <cpp:class>

2490 <cpp:class> provides a mechanism for defining an alternate C++ class name for a WSDL construct.

2491

2492 Format:

```
2493 <cpp:class name="xsd:string"/>
```

2494 where:

- 2495 • **class/@name : NCName (1..1)** – specifies the name of the C++ class associated with this WSDL
2496 element.

2497

2498 Applicable WSDL element(s):

- 2499 • wsdl:portType
- 2500 • wsdl:fault

2501

2502 A <cpp:bindings/> element MUST NOT have more than one <cpp:class/> child element. [CPPD0002]

2503

2504 Example:

2505 Input WSDL file:

```
2506 <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"  
2507     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"  
2508     xmlns:tns="http://www.example.org/"  
2509     xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"  
2510     targetNamespace="http://www.example.org/">  
2511  
2512     <portType name="StockQuote">  
2513         <cpp:bindings>  
2514             <cpp:class name="StockQuoteService"/>  
2515         </cpp:bindings>  
2516     </portType>  
2517 </definitions>
```

2518

2519 Generated C++ file:

```
2520 // @WebService(name="StockQuote", targetNamespace="http://www.example.org/"  
2521 //     serviceName="StockQuoteService")
```

```
2522 class StockQuoteService {
2523 };
2524
```

2525 **D.3 <cpp:enableWrapperStyle>**

2526 <cpp:enableWrapperStyle> indicates whether or not the wrapper style for messages should be applied,
2527 when otherwise applicable. If false, the wrapper style will never be applied.

2528

2529 **Format:**

```
2530 <cpp:enableWrapperStyle>value</cpp:enableWrapperStyle>
```

2531 where:

- 2532 • **enableWrapperStyle/text() : boolean (1..1)** – specifies whether wrapper style should be enabled or
2533 disabled for this element and any of its children. The default value is “true”.

2534

2535 **Applicable WSDL element(s):**

- 2536 • wsdl:definitions
- 2537 • wsdl:portType – overrides a binding applied to wsdl:definitions
- 2538 • wsdl:portType/wsdl:operation – overrides a binding applied to wsdl:definitions or the enclosing
2539 wsdl:portType

2540

2541 <cpp:bindings/> element MUST NOT have more than one <cpp:enableWrapperStyle/> child element.

2542 **[CPPD0003]**

2543

2544 **Example:**

2545 **Input WSDL file:**

```
2546 <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
2547 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
2548 xmlns:tns="http://www.example.org/"
2549 xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
2550 targetNamespace="http://www.example.org/"
2551
2552 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
2553 xmlns:tns="http://www.example.org/"
2554 attributeFormDefault="unqualified"
2555 elementFormDefault="unqualified"
2556 targetNamespace="http://www.example.org/"
2557 <xs:element name="GetLastTradePrice" type="tns:GetLastTradePrice"/>
2558 <xs:element name="GetLastTradePriceResponse"
2559 type="tns:GetLastTradePriceResponse"/>
2560 <xs:complexType name="GetLastTradePrice">
2561 <xs:sequence>
2562 <xs:element name="tickerSymbol" type="xs:string"/>
2563 </xs:sequence>
2564 </xs:complexType>
2565 <xs:complexType name="GetLastTradePriceResponse">
2566 <xs:sequence>
2567 <xs:element name="return" type="xs:float"/>
2568 </xs:sequence>
2569 </xs:complexType>
2570 </xs:schema>
2571
2572 < message name="GetLastTradePrice">
2573 <part name="parameters" element="tns:GetLastTradePrice">
2574 </part>
```

```

2575 </message>
2576
2577 < message name="GetLastTradePriceResponse">
2578   <part name="parameters" element="tns:GetLastTradePriceResponse">
2579   </part>
2580 </ message>
2581
2582 <portType name="StockQuote">
2583   <cpp:bindings>
2584     <cpp:class name="StockQuoteService"/>
2585     <cpp:enableWrapperStyle>>false</cpp:enableWrapperStyle>
2586   <cpp:bindings>
2587     <operation name="GetLastTradePrice">
2588       <cpp:bindings>
2589         <cpp:memberFunction name="getLastTradePrice"/>
2590       </cpp:bindings>
2591       <input name="GetLastTradePrice" message="tns:GetLastTradePrice">
2592       </input>
2593       <output name="GetLastTradePriceResponse"
2594         message="tns:GetLastTradePriceResponse">
2595       </output>
2596     </operation>
2597   </portType>
2598 </definitions>

```

2599

Generated C++ file:

```

2600
2601 // @WebService(name="StockQuote", targetNamespace="http://www.example.org/"
2602 //   serviceName="StockQuoteService")
2603 class StockQuoteService {
2604
2605   // @WebFunction(operationName="GetLastTradePrice",
2606   //   action="urn:GetLastTradePrice")
2607   commonj::sdo::DataObjectPtr
2608   getLastTradePrice(commonj::sdo::DataObjectPtr parameters);
2609 };

```

2610

2611 D.4 <cpp:namespace>

2612 <cpp:namespace> specifies the name of the C++ namespace that the associated WSDL element (and
2613 any of it's children) should be created in.

2614

Format:

```
2615 <cpp:namespace name="namespaceURI"/>
```

2617 where:

- 2618 • **namespace/@name : anyURI (1..1)** – specifies the name of the C++ namespace associated with
2619 this WSDL element.

2620

Applicable WSDL element(s):

- 2622 • wsdl:definitions

2623

2624 A <cpp:bindings/> element MUST NOT have more than one <cpp:namespace/> child element.

2625 [CPPD0004]

2626

Example:

2627

2628 Input WSDL file:

```
2629 <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
2630             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
2631             xmlns:tns="http://www.example.org/"
2632             xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
2633             targetNamespace="http://www.example.org/">
2634   <cpp:bindings>
2635     <cpp:namespace name="stock"/>
2636   </cpp:bindings>
2637
2638   <portType name="StockQuote">
2639     <cpp:bindings>
2640       <cpp:class name="StockQuoteService"/>
2641     </cpp:bindings>
2642   </portType>
2643 </definitions>
```

2644

2645 Generated C++ file:

```
2646 namespace stock
2647 {
2648   // @WebService(name="StockQuote", targetNamespace="http://www.example.org/"
2649   //           serviceName="StockQuoteService")
2650   // @WebService(name="StockQuote",
2651   class StockQuoteService {
2652   };
2653 }
```

2654

2655 D.5 <cpp:memberFunction>

2656 <cpp:memberFunction> specifies the name of the C++ member function that the associated WSDL
2657 operation should be associated with.

2658

2659 **Format:**

```
2660 <cpp:memberFunction name="myFunction"/>
```

2661 where:

- 2662 • **memberFunction/@name : NCName (1..1)** – specifies the name of the C++ member function
2663 associated with this WSDL operation.

2664

2665 **Applicable WSDL element(s):**

- 2666 • wsd:portType/wsd:operation

2667

2668 A <cpp:bindings/> element MUST NOT have more than one <cpp:memberFunction/> child element.

2669 [CPPD0005]

2670

2671 **Example:**

2672 Input WSDL file:

```
2673 <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
2674             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
2675             xmlns:tns="http://www.example.org/"
2676             xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
2677             targetNamespace="http://www.example.org/">
2678
2679   <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
```

```

2680     xmlns:tns="http://www.example.org/"
2681     attributeFormDefault="unqualified"
2682     elementFormDefault="unqualified"
2683     targetNamespace="http://www.example.org/">
2684 <xs:element name="GetLastTradePrice" type="tns:GetLastTradePrice"/>
2685 <xs:element name="GetLastTradePriceResponse"
2686     type="tns:GetLastTradePriceResponse"/>
2687 <xs:complexType name="GetLastTradePrice">
2688     <xs:sequence>
2689         <xs:element name="tickerSymbol" type="xs:string"/>
2690     </xs:sequence>
2691 </xs:complexType>
2692 <xs:complexType name="GetLastTradePriceResponse">
2693     <xs:sequence>
2694         <xs:element name="return" type="xs:float"/>
2695     </xs:sequence>
2696 </xs:complexType>
2697 </xs:schema>
2698
2699 < message name="GetLastTradePrice">
2700     <part name="parameters" element="tns:GetLastTradePrice">
2701     </part>
2702 </message>
2703
2704 < message name="GetLastTradePriceResponse">
2705     <part name="parameters" element="tns:GetLastTradePriceResponse">
2706     </part>
2707 </ message>
2708
2709 <portType name="StockQuote">
2710     <cpp:bindings>
2711         <cpp:class name="StockQuoteService"/>
2712     </cpp:bindings>
2713     <operation name="GetLastTradePrice">
2714         <cpp:bindings>
2715             <cpp:memberFunction name="getTradePrice"/>
2716         </cpp:bindings>
2717         <input name="GetLastTradePrice" message="tns:GetLastTradePrice">
2718         </input>
2719         <output name="GetLastTradePriceResponse"
2720             message="tns:GetLastTradePriceResponse">
2721         </output>
2722     </operation>
2723 </portType>
2724 </definitions>

```

2725

Generated C++ file:

```

2727 // @WebService(name="StockQuote", targetNamespace="http://www.example.org/"
2728 //     serviceName="StockQuoteService")
2729 class StockQuoteService {
2730
2731     // @WebFunction(operationName="GetLastTradePrice",
2732     //     action="urn:GetLastTradePrice")
2733     float getTradePrice(const std::string& tickerSymbol);
2734 };

```

2735

2736 D.6 <cpp:parameter>

2737 <cpp:parameter> specifies the name of the C++ member function parameter associated with a specific
2738 WSDL message part or wrapper child element.

2739

2740 **Format:**

```
2741 <cpp:parameter name="CPPParameter" part="WSDLPart"  
2742 childElementName="WSDLElement" type="CPPTYPE"/>
```

2743 where:

- 2744 • **parameter/@name : NCName (1..1)** – specifies the name of the C++ member function parameter
2745 associated with this WSDL operation. “return” is used to denote the return value.
- 2746 • **parameter/@part : string (1..1)** - an XPath expression identifying the wsdl:part of a wsdl:message.
- 2747 • **parameter/@childElementName : QName (1..1)** – specifies the qualified name of a child element of
2748 the global element identified by parameter/@part.
- 2749 • **type : NCName (0..1)** – specifies the type of the parameter or struct member or return type. The
2750 value of the type property of a @WebResult annotation MUST be one of the simpleTypes defined in
2751 namespace http://www.w3.org/2001/XMLSchema. [CPPD0006] The default type is determined by the
2752 mapping defined in Simple Content Binding.

2753

2754 **Applicable WSDL element(s):**

- 2755 • wsdl:portType/wsdl:operation

2756

2757 **Example:**

2758 **Input WSDL file:**

```
2759 <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"  
2760 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"  
2761 xmlns:tns="http://www.example.org/"  
2762 xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"  
2763 targetNamespace="http://www.example.org/">  
2764  
2765 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"  
2766 xmlns:tns="http://www.example.org/"  
2767 attributeFormDefault="unqualified"  
2768 elementFormDefault="unqualified"  
2769 targetNamespace="http://www.example.org/">  
2770 <xs:element name="GetLastTradePrice" type="tns:GetLastTradePrice"/>  
2771 <xs:element name="GetLastTradePriceResponse"  
2772 type="tns:GetLastTradePriceResponse"/>  
2773 <xs:complexType name="GetLastTradePrice">  
2774 <xs:sequence>  
2775 <xs:element name="symbol" type="xs:string"/>  
2776 </xs:sequence>  
2777 </xs:complexType>  
2778 <xs:complexType name="GetLastTradePriceResponse">  
2779 <xs:sequence>  
2780 <xs:element name="return" type="xs:float"/>  
2781 </xs:sequence>  
2782 </xs:complexType>  
2783 </xs:schema>  
2784  
2785 < message name="GetLastTradePrice">  
2786 <part name="parameters" element="tns:GetLastTradePrice">  
2787 </part>  
2788 </message>  
2789  
2790 < message name="GetLastTradePriceResponse">  
2791 <part name="parameters" element="tns:GetLastTradePriceResponse">  
2792 </part>  
2793 </ message>  
2794
```

```

2795 <portType name="StockQuote">
2796   <cpp:bindings>
2797     <cpp:class name="StockQuoteService"/>
2798   </cpp:bindings>
2799   <operation name="GetLastTradePrice">
2800     <cpp:bindings>
2801       <cpp:memberFunction name="getLastTradePrice"/>
2802       <cpp:parameter name="tickerSymbol"
2803         part="tns:GetLastTradePrice/parameter"
2804         childElementName="symbol"/>
2805     </cpp:bindings>
2806     <input name="GetLastTradePrice" message="tns:GetLastTradePrice">
2807     </input>
2808     <output name="GetLastTradePriceResponse"
2809       message="tns:GetLastTradePriceResponse">
2810     </output>
2811   </operation>
2812 </portType>
2813
2814 <binding name="StockQuoteServiceSoapBinding">
2815   <soap:binding style="document"
2816     transport="http://schemas.xmlsoap.org/soap/http"/>
2817   <wsdl:operation name="GetLastTradePrice">
2818     <soap:operation soapAction="urn:GetLastTradePrice" style="document"/>
2819     <wsdl:input name="GetLastTradePrice">
2820       <soap:body use="literal"/>
2821     </wsdl:input>
2822     <wsdl:output name="GetLastTradePriceResponse">
2823       <soap:body use="literal"/>
2824     </wsdl:output>
2825   </wsdl:operation>
2826 </binding>
2827
2828 <service name="StockQuoteService">
2829   <port name="StockQuotePort" binding="tns:StockQuoteServiceSoapBinding">
2830     <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
2831   </port>
2832 </service>
2833 </definitions>

```

2834

Generated C++ file:

```

2836 // @WebService(name="StockQuote", targetNamespace="http://www.example.org/"
2837 //   serviceName="StockQuoteService")
2838 class StockQuoteService {
2839
2840   // @WebFunction(operationName="GetLastTradePrice",
2841   //   action="urn:GetLastTradePrice")
2842   // @WebParam(paramName="tickerSymbol", name="symbol")
2843   float getLastTradePrice(const std::string& tickerSymbol);
2844 };

```

2845

2846 D.7 JAX-WS WSDL Extensions

2847 An SCA implementation MAY support the reading and interpretation of JAX-WS defined WSDL
 2848 extensions; however it MUST give precedence to the corresponding SCA WSDL extension if present.

2849 **[CPPD0007]** The following is a list of JAX-WS WSDL extensions that MAY be recognized, and their
 2850 corresponding SCA WSDL extension.

2851

JAX-WS Extension	SCA Extension
jaxws:bindings	cpp:bindings
jaxws:class	cpp:class
jaxws:method	cpp:memberFunction
jaxws:parameter	cpp:parameter
jaxws:enableWrapperStyle	cpp:enableWrapperStyle

2852 D.8 WSDL Extensions Schema

2853 The normative schema defining the WSDL extensions for C++ is located at:

- 2854 • <http://docs.oasis-open.org/opencsa/sca-c-cpp/cpp/200901/sca-wsdlex-c-1.1-cd02.xsd>

2855

2856 The following copy is provided for reference.

2857

```

2858 <?xml version="1.0" encoding="UTF-8"?>
2859 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2860         targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca-c-
2861 cpp/cpp/200901"
2862         xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
2863         xmlns:xsd="http://www.w3.org/2001/XMLSchema"
2864         elementFormDefault="qualified">
2865
2866     <element name="bindings" type="cpp:BindingsType" />
2867     <complexType name="BindingsType">
2868         <choice minOccurs="0" maxOccurs="unbounded">
2869             <element ref="cpp:namespace" />
2870             <element ref="cpp:class" />
2871             <element ref="cpp:enableWrapperStyle" />
2872             <element ref="cpp:memberFunction" />
2873             <element ref="cpp:parameter" />
2874         </choice>
2875     </complexType>
2876
2877     <element name="namespace" type="cpp:NamespaceType" />
2878     <complexType name="NamespaceType">
2879         <attribute name="name" type="xsd:anyURI" use="required" />
2880     </complexType>
2881
2882     <element name="class" type="cpp:ClassType" />
2883     <complexType name="ClassType">
2884         <attribute name="name" type="xsd:NCName" use="required" />
2885     </complexType>
2886
2887     <element name="memberFunction" type="cpp:MemberFunctionType" />
2888     <complexType name="MemberFunctionType">
2889         <attribute name="name" type="xsd:NCName" use="required" />
2890     </complexType>
2891
2892     <element name="parameter" type="cpp:ParameterType" />
2893     <complexType name="ParameterType">
2894         <attribute name="part" type="xsd:string" use="required" />
2895         <attribute name="childElementName" type="xsd:QName"
2896             use="required" />
2897         <attribute name="name" type="xsd:NCName" use="required" />
2898         <attribute name="type" type="xsd:string" use="optional" />

```

2899
2900
2901
2902

```
        </complexType>  
        <element name="enableWrapperStyle" type="xsd:boolean" />  
</schema>
```

2903 E XML Schemas

2904 Three XML schemas are defined to support the use of C++ for implementation and definition of
2905 interfaces.

2906 The normative schemas are located at:

- 2907 • <http://docs.oasis-open.org/opencsa/sca/2007/12/sca-interface-cpp-1.1-cd02.xsd>
- 2908 • <http://docs.oasis-open.org/opencsa/sca/2007/12/sca-implementation-cpp-1.1-cd02.xsd>
- 2909 • <http://docs.oasis-open.org/opencsa/sca/2007/12/sca-contribution-cpp-1.1-cd02.xsd>

2910

2911 The following copies are provided for reference.

2912 E.1 sca-interface-cpp-1.1.xsd

```
2913 <?xml version="1.0" encoding="UTF-8"?>
2914 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2915         targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2916         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2917         elementFormDefault="qualified">
2918
2919     <include schemaLocation="sca-core.xsd"/>
2920
2921     <element name="interface.cpp" type="sca:CPPInterface"
2922           substitutionGroup="sca:interface"/>
2923
2924     <complexType name="CPPInterface">
2925         <complexContent>
2926             <extension base="sca:Interface">
2927                 <sequence>
2928                     <element name="function" type="sca:CPPFunction"
2929                           minOccurs="0" maxOccurs="unbounded" />
2930                     <element name="callbackFunction" type="sca:CPPFunction"
2931                           minOccurs="0" maxOccurs="unbounded" />
2932                     <any namespace="##other" processContents="lax"
2933                           minOccurs="0" maxOccurs="unbounded"/>
2934                 </sequence>
2935                 <attribute name="header" type="string" use="required"/>
2936                 <attribute name="class" type="Name" use="required"/>
2937                 <attribute name="callbackHeader" type="string" use="optional"/>
2938                 <attribute name="callbackClass" type="Name" use="optional"/>
2939                 <attribute name="remotable" type="boolean" use="optional"/>
2940                 <anyAttribute namespace="##other" processContents="lax"/>
2941             </extension>
2942         </complexContent>
2943     </complexType>
2944
2945     <complexType name="CPPFunction">
2946         <attribute name="name" type="NCName" use="required"/>
2947         <attribute name="oneWay" type="boolean" use="optional"/>
2948         <anyAttribute namespace="##other" processContents="lax"/>
2949     </complexType>
2950
2951 </schema>
```

2952 E.2 sca-implementation-cpp-1.1.xsd

```
2953 <?xml version="1.0" encoding="UTF-8"?>
```

```

2954 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2955         targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2956         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2957         elementFormDefault="qualified">
2958
2959     <include schemaLocation="sca-core.xsd"/>
2960
2961     <element name="implementation.cpp" type="sca:CPPImplementation"
2962           substitutionGroup="sca:implementation" />
2963     <complexType name="CPPImplementation">
2964       <complexContent>
2965         <extension base="sca:Implementation">
2966           <sequence>
2967             <element name="function" type="sca:CPPImplementationFunction"
2968               minOccurs="0" maxOccurs="unbounded" />
2969             <any namespace="##other" processContents="lax"
2970               minOccurs="0" maxOccurs="unbounded"/>
2971           </sequence>
2972           <attribute name="library" type="NCName" use="required"/>
2973           <attribute name="header" type="NCName" use="required"/>
2974           <attribute name="path" type="string" use="optional"/>
2975           <attribute name="class" type="Name" use="optional"/>
2976           <attribute name="componentType" type="string" use="optional"/>
2977           <attribute name="scope" type="sca:CPPImplementationScope"
2978             use="optional"/>
2979           <attribute name="eagerInit" type="boolean" use="optional"/>
2980           <attribute name="allowsPassByReference" type="boolean"
2981             use="optional"/>
2982           <anyAttribute namespace="##other" processContents="lax"/>
2983         </extension>
2984       </complexContent>
2985     </complexType>
2986
2987     <simpleType name="CPPImplementationScope">
2988       <restriction base="string">
2989         <enumeration value="stateless"/>
2990         <enumeration value="composite"/>
2991       </restriction>
2992     </simpleType>
2993
2994     <complexType name="CPPImplementationFunction">
2995       <attribute name="name" type="NCName" use="required"/>
2996       <attribute name="allowsPassByReference" type="boolean"
2997         use="optional"/>
2998       <anyAttribute namespace="##other" processContents="lax"/>
2999     </complexType>
3000
3001 </schema>

```

3002 **E.3 sca-contribution-cpp-1.1.xsd**

```

3003 <?xml version="1.0" encoding="UTF-8"?>
3004 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3005         targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3006         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3007         elementFormDefault="qualified">
3008
3009     <include schemaLocation="sca-contributions.xsd"/>
3010
3011     <element name="export.cpp" type="sca:CPPEExport"
3012           substitutionGroup="sca:Export"/>
3013
3014     <complexType name="CPPEExport">

```

```
3015     <complexContent>
3016         <attribute name="name" type="QName" use="required"/>
3017         <attribute name="path" type="string" use="optional"/>
3018     </complexContent>
3019 </complexType>
3020
3021 <element name="import.cpp" type="sca:CPPImport"
3022     substitutionGroup="sca:Import"/>
3023
3024 <complexType name="CPPImport">
3025     <complexContent>
3026         <attribute name="name" type="QName" use="required"/>
3027         <attribute name="location" type="string" use="required"/>
3028     </complexContent>
3029 </complexType>
3030
3031 </schema>
```

3032

F Conformance Items

3033 This section contains a list of conformance items for the SCA C++ Client and Implementation Model
3034 specification.

Conformance ID	Description
[CPP20001]	A C++ implementation MUST implement all of the operation(s) of the service interface(s) of its componentType.
[CPP20002]	A C++ implementation of a remotable service that allows pass by reference MUST NOT alter its input data during or after the invocation, and MUST NOT modify return data after invocation.
[CPP20003]	An SCA runtime MUST support these scopes; stateless and composite . Additional scopes MAY be provided by SCA runtimes.
[CPP20005]	If the header file identified by the <code>@header</code> attribute of an <code><interface.cpp/></code> element contains more than one class, then the <code>@class</code> attribute MUST be specified for the <code><interface.cpp/></code> element.
[CPP20006]	If the header file identified by the <code>@callbackHeader</code> attribute of an <code><interface.cpp/></code> element contains more than one class, then the <code>@callbackClass</code> attribute MUST be specified for the <code><interface.cpp/></code> element.
[CPP20007]	The <code>@name</code> attribute of a <code><function/></code> child element of a <code><interface.cpp/></code> MUST be unique amongst the <code><function/></code> elements of that <code><interface.cpp/></code> .
[CPP20008]	The <code>@name</code> attribute of a <code><callbackFunction/></code> child element of a <code><interface.cpp/></code> MUST be unique amongst the <code><callbackFunction/></code> elements of that <code><interface.cpp/></code> .
[CPP20009]	The name of the componentType file for a C++ implementation MUST match the class name (excluding any namespace definition) of the implementations as defined by the <code>@class</code> attribute of the <code><implementation.cpp/></code> element.
[CPP20010]	The <code>@name</code> attribute of a <code><function/></code> child element of a <code><implementation.cpp/></code> MUST be unique amongst the <code><function/></code> elements of that <code><implementation.cpp/></code> .
[CPP20011]	A C++ implementation class MUST be default constructable by the SCA runtime to instantiate the component.
[CPP20012]	An SCA runtime MUST ensure that a stateless scoped implementation instance object is only ever dispatched on one thread at any one time. In addition, within the SCA lifecycle of an instance, an SCA runtime MUST only make a single invocation of one business method.
[CPP20013]	An SCA runtime MAY run multiple threads in a single composite scoped implementation instance object and it MUST NOT perform any synchronization.
[CPP40001]	A member function marked as <code>oneWay="true"</code> is considered non-blocking and the SCA runtime MAY use a binding that buffers the requests to the member function and sends them at some time after they are made.
[CPP60001]	An SCA runtime MUST implement Reference Counting Pointers, the ComponentContext, Service Reference and SCAExceptions classes.
[CPP70001]	The <code>@name</code> attribute of a <code><export.cpp/></code> element MUST be unique amongst the

	<export.cpp/> elements in a domain.
[CPP70002]	The @name attribute of a <import.cpp/> child element of a <contribution/> MUST be unique amongst the <import.cpp/> elements in of that contribution.
[CPP90001]	A C++ header file used to define an interface MUST: <ul style="list-style-type: none"> • Declare at least one class with: <ul style="list-style-type: none"> – At least one public member function. – All public member functions must be pure virtual (virtual with no implementation)
[CPP90002]	A C++ header file used to define an interface MUST NOT use the following constructs: <ul style="list-style-type: none"> • Macros • Inline member functions • Friend classes
[CPP100001]	A WSDLfile might define a namespace using the <sca:namespace> WSDL extension, otherwise all C++ classes MUST be placed in a default namespace as determined by the implementation. Implementations SHOULD provide a mechanism for overriding the default namespace.
[CPP100002]	If multiple operations within the same portType indicate that they throw faults that reference the same global element, an SCA implementation MUST generate a single C++ exception class with each C++ member function referencing this class in its @WebThrows annotation.
[CPP100003]	<ul style="list-style-type: none"> • For unwrapped messages, an SCA implementation MUST map: <ul style="list-style-type: none"> – in - the message part to a member function parameter, passed by const-reference. – out - the message part to a member function parameter, passed by reference, or to the member function return type, returned by-value. – in/out - the message part to a member function parameter, passed by reference.
[CPP100004]	<ul style="list-style-type: none"> • For wrapped messages, an SCA implementation MUST map: <ul style="list-style-type: none"> – in - the wrapper child to a member function parameter, passed by const-reference. – out - the wrapper child to a member function parameter, passed by reference, or to the member function return type, returned by-value. – in/out - the wrapper child to a member function parameter, passed by reference.
[CPP100005]	An SCA implementation SHOULD provide a mechanism for overriding the default targetNamespace.
[CPP100006]	An SCA implementation MUST map a method's return type as an out parameter, a parameter passed by-reference or by-pointer as an in/out parameter, and all other parameters, including those passed by-const-reference as in parameters.
[CPP100007]	An SCA implementation must ensure each class that is referenced from an @WebThrows annotation MUST itself have a @WebFault annotation that associates the fault with a particular global element that will be associated with the fault message.

[CPPA0001]	An SCA implementation MAY support source file annotations. If annotations are supported by an implementation, the annotations defined here MUST be supported and MUST be mapped to SCDL as described. The SCA runtime MUST only process the SCDL files and not the annotations.
[CPPA0002]	If multiple annotations apply to a program element, all of the annotations SHOULD be in the same comment block.
[CPPA0003]	An SCA implementation MUST treat a class with an @WebService annotation specified.
[CPPC0001]	An SCA implementation MAY support source file annotations for WSDL. If annotations are supported by an implementation, the annotations defined here MUST be supported and MUST be mapped to WSDL as described.
[CPPC0002]	An SCA implementation MUST treat any instance of a @Interface annotation and without an explicit @WebService annotation as if a @WebService annotation with no parameters was specified.
[CPPC0003]	If a @WebService does not have a portName element, an SCA implementation MUST use the value associated with the name element, suffixed with "Port".
[CPPC0004]	Only named parameters MAY be referenced by an @WebParam annotation.
[CPPC0005]	The value of the type property of a @WebParam annotation MUST be one of the simpleTypes defined in namespace http://www.w3.org/2001/XMLSchema .
[CPPC0006]	The value of the type property of a @WebResult annotation MUST be one of the simpleTypes defined in namespace http://www.w3.org/2001/XMLSchema .
[CPPC0007]	A C++ class with a @WebFault annotation MUST provide a constructor that takes two parameters, a std::string and a type representing the fault information. Additionally, the class MUST provide a const member function "getFaultInfo" that takes no parameters, and returns the same type as defined in the constructor.
[CPPC0008]	A C++ class that is listed in a @WebThrows annotation MUST itself have a @WebFault annotation.
[CPPD0001]	An SCA implementation MAY support these WSDL extensions. If these extensions are supported by an implementation, all the extensions defined here MUST be supported and MUST be mapped to C++ as described.
[CPPD0002]	A <cpp:bindings/> element MUST NOT have more than one <cpp:class/> child element.
[CPPD0003]	<cpp:bindings/> element MUST NOT have more than one <cpp:enableWrapperStyle/> child element.
[CPPD0004]	A <cpp:bindings/> element MUST NOT have more than one <cpp:namespace/> child element.
[CPPD0005]	A <cpp:bindings/> element MUST NOT have more than one <cpp:memberFunction/> child element.
[CPPD0006]	The @type attribute of a <parameter/> element MUST be a valid C++ type.
[CPPD0007]	An SCA implementation MAY support the reading and interpretation of JAX-WS defined WSDL extensions; however it MUST give precedence to the corresponding SCA WSDL extension if present.

3035 **F.1 JAX-WS Conformance**

3036 The JAX-WS 2.1 specification [JAXWS21] defines conformance statements for various requirements
 3037 defined by that specification. The following table outlines those conformance statements, and describes
 3038 whether the conformance statement applies to the WSDL binding described in this specification.

Section	Conformance Statement	Notes	Conformance ID
2	WSDL 1.1 support	[A]	[CPPF0001]
2	Customization required	[CPPD0001] The reference to the JAX-WS binding language should be treated as a reference to the C++ WSDL extensions defined in section WSDL C++ Mapping Extensions.	
2	Annotations on generated classes		[CPPF0002]
2.1	WSDL and XML Schema import directives		[CPPF0003]
2.1.1	Optional WSDL extensions		[CPPF0004]
2.2	SEI naming		[CPPF0005]
2.2	javax.jws.WebService required	[B] References to javax.jws.WebService in the conformance statement should be treated as the C++ annotation @WebService.	[CPPF0006]
2.3	Method naming		[CPPF0007]
2.3	javax.jws.WebMethod required	[A], [B] References to javax.jws.WebMethod in the conformance statement should be treated as the C++ annotation @WebFunction.	[CPPF0008]
2.3	Transmission primitive support		[CPPF0009]
2.3	Using javax.jws.OneWay	[A], [B] References to javax.jws.OneWay in the conformance statement should be treated as the C++ annotation @OneWay.	[CPPF0010]
2.3.1	Using javax.jws.SOAPBinding	[A], [B] References to javax.jws.SOAPBinding in the conformance statement should be treated as the C++ annotation @SOAPBinding.	[CPPF0011]
2.3.1	Using javax.jws.WebParam	[A], [B] References to javax.jws.WebParam in the conformance statement should be treated as the C++ annotation @WebParam.	[CPPF0012]
2.3.1	Using javax.jws.WebResult	[A], [B] References to javax.jws.WebResult in the	[CPPF0013]

		conformance statement should be treated as the C++ annotation @WebResult.	
2.3.1.1	Non-wrapped parameter naming		[CPPF0014]
2.3.1.2	Default mapping mode		[CPPF0015]
2.3.1.2	Disabling wrapper style	[B] References to javax:enableWrapperStyle in the conformance statement should be treated as the WSDL extension cpp:enableWrapperStyle.	[CPPF0016]
2.3.1.2	Wrapped parameter naming		[CPPF0017]
2.3.1.2	Parameter name clash	[A]	[CPPF0018]
2.5	javax.xml.ws.WebFault required	[B] References to javax.jws.WebFault in the conformance statement should be treated as the C++ annotation @WebFault.	[CPPF0019]
2.5	Exception naming		[CPPF0020]
2.5	Fault equivalence	[A]	[CPPF0021]
2.6	Required WSDL extensions	MIME Binding not required	[CPPF0022]
2.6.1	Unbound message parts	[A]	[CPPF0023]
2.6.2.1	Duplicate headers in binding		[CPPF0024]
2.6.2.1	Duplicate headers in message		[CPPF0025]
3	WSDL 1.1 support	[A]	[CPPF0026]
3	Standard annotations	[A] [CPPC0001]	
3.1	Java identifier mapping	[A]	[CPPF0027]
3.1.1	Method name disambiguation	[A] References to javax.jws.WebMethod in the conformance statement should be treated as the C++ annotation @WebFunction.	[CPPF0028]
3.2	WSDL and XML Schema import directives		[CPPF0029]
3.4	portType naming		[CPPF0030]
3.5	Operation naming		[CPPF0031]
3.5.1	One-way mapping	[B] References to javax.jws.OneWay in the conformance statement should be treated	[CPPF0032]

as the C++ annotation @OneWay.

3.5.1	One-way mapping errors		[CPPF0033]
3.6.1	Parameter classification		[CPPF0034]
3.6.1	Parameter naming		[CPPF0035]
3.6.1	Result naming		[CPPF0036]
3.6.1	Header mapping of parameters and results	References to javax.jws.WebParam in the conformance statement should be treated as the C++ annotation @WebParam. References to javax.jws.WebResult in the conformance statement should be treated as the C++ annotation @WebResult.	[CPPF0037]
3.7	Exception naming	[A] References to javax.jws.WebFault in the conformance statement should be treated as the C++ annotation @WebFault.	[CPPF0038]
3.8	Binding selection	References to the BindingType annotation should be treated as references to SOAP related intents defined by [POLICY] .	[CPPF0039]
3.10	SOAP binding support	[A]	[CPPF0040]
3.10.1	SOAP binding style required		[CPPF0041]
3.11	Port selection		[CPPF0042]
3.11	Port binding	References to the BindingType annotation should be treated as references to SOAP related intents defined by [POLICY] .	[CPPF0043]

3039 [A] All references to Java in the conformance statement should be treated as C++.

3040 [B] Annotation generation is only required if annotations are supported by an SCA implementation.

3041 F.1.1 Ignored Conformance Statements

Section	Conformance Statement	Notes
2.1	Definitions mapping	
2.2	javax.xml.bind.XmlSeeAlso required	
2.3.1	use of JAXB annotations	
2.3.1.2	Using javax.xml.ws.RequestWrapper	
2.3.1.2	Using javax.xml.ws.ResponseWrapper	
2.3.3	Use of Holder	
2.3.4	Asynchronous mapping required	
2.3.4	Asynchronous mapping option	
2.3.4.2	Asynchronous method naming	

2.3.4.2	Asynchronous parameter naming	
2.3.4.2	Failed method invocation	
2.3.4.4	Response bean naming	
2.3.4.5	Asynchronous fault reporting	
2.3.4.5	Asynchronous fault cause	
2.4	JAXB class mapping	
2.4	JAXB customization use	
2.4	JAXB customization clash	
2.4.1	javax.xml.ws.wsaddressing.W3CEndpointReference	
2.6.3.1	Use of MIME type information	
2.6.3.1	MIME type mismatch	
2.6.3.1	MIME part identification	
2.7	Service superclass required	
2.7	Service class naming	
2.7	javax.xml.ws.WebServiceClient required	
2.7	Default constructor required	
2.7	2 argument constructor required	
2.7	Failed getPort Method	
2.7	javax.xml.ws.WebEndpoint required	
3.2	Package name mapping	
3.3	Class mapping	
3.4.1	Inheritance flattening	
3.4.1	Inherited interface mapping	
3.6	use of JAXB annotations	
3.6.2.1	Default wrapper bean names	
3.6.2.1	Default wrapper bean package	
3.6.2.3	Null Values in rpc/literal	
3.7	java.lang.RuntimeExceptions and java.rmi.RemoteExceptions	
3.7	Fault bean name clash	
3.11	Service creation	

3042 G Migration

3043 To aid migration of an implementation or clients using an implementation based the version of the Service
3044 Component Architecture for C++ defined in [OSOA SCA C++ Client and Implementation V1.00](#), this
3045 appendix identifies the relevant changes to APIs, annotations, or behavior defined in V1.00.

3046 G.1 Method child elements of `interface.cpp` and `implementation.cpp`

3047 The `<method/>` child element of `<interface.cpp/>` and the `<method/>` child element of
3048 `<implementation.cpp/>` have both been renamed to `<function/>` to be consistent with C++ terminology.

3049

H Acknowledgements

3050 The following individuals have participated in the creation of this specification and are gratefully
3051 acknowledged:

3052 **Participants:**

3053 Andrew Borley, IBM

3054 Bryan Aupperle, IBM

3055 David Haney, Rogue Wave Software

3056 Jeff Mischkin, Oracle

3057 Mike Edwards, IBM

3058 Pete Robbins, IBM

3060

J Revision History

3061

[optional; should not be included in OASIS Standards]

3062

Revision	Date	Editor	Changes Made
			•

3063