



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

Committee Draft 05 / Public Review Draft 03

4 March 2010

Specification URIs:

This Version:

<http://docs.oasis-open.org/opencsa/sca-c-cpp/sca-cppcni-1.1-spec-cd05.html>
<http://docs.oasis-open.org/opencsa/sca-c-cpp/sca-cppcni-1.1-spec-cd05.doc>
<http://docs.oasis-open.org/opencsa/sca-c-cpp/sca-cppcni-1.1-spec-cd05.pdf> (Authoritative)

Previous Version:

<http://docs.oasis-open.org/opencsa/sca-c-cpp/sca-cppcni-1.1-spec-cd04.html>
<http://docs.oasis-open.org/opencsa/sca-c-cpp/sca-cppcni-1.1-spec-cd04.doc>
<http://docs.oasis-open.org/opencsa/sca-c-cpp/sca-cppcni-1.1-spec-cd04.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

Editors:

Bryan Aupperle, IBM
David Haney
Pete Robbins, IBM

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/200912>
<http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901>

C++ Artifacts:

<http://docs.oasis-open.org/opencsa/sca-c-cpp/sca-cppcni-1.1-apis-cd05.zip>

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, 2010. 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	Conventions	9
1.3.1	Naming Conventions	9
1.3.2	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	AllowsPassByReference	12
2.1.3	Implementing a Local Service	12
2.2	Component Implementation Scopes	13
2.2.1	Stateless Scope	13
2.2.2	Composite Scope	13
2.3	Implementing a Configuration Property	13
2.4	Component Type and Component	14
2.4.1	Interface.cpp	15
2.4.2	Function and CallbackFunction	16
2.4.3	Implementation.cpp	17
2.4.4	Implementation Function	18
2.5	Instantiation.....	19
3	Basic Client Model.....	20
3.1	Accessing Services from Component Implementations	20
3.2	Interface Proxies	21
3.3	Accessing Services from non-SCA Component Implementations	23
3.4	Calling Service Operations	23
3.5	Long Running Request-Response Operations.....	23
3.5.1	Response Callback	25
3.5.2	Response Polling	26
3.5.3	Synchronous Response Access.....	26
3.5.4	Response Class	27
4	Asynchronous Programming	29
4.1	Non-blocking Calls	29
4.2	Callbacks	29
4.2.1	Using Callbacks.....	30
4.2.2	Callback Instance Management	31
4.2.3	Implementing Multiple Bidirectional Interfaces	32
5	Error Handling	33
6	C++ API	34
6.1	Reference Counting Pointers.....	34
6.1.1	operator*	34
6.1.2	operator->	35
6.1.3	operator void*	35

6.1.4 operator!	35
6.1.5 constCast.....	35
6.1.6 dynamicCast.....	35
6.1.7 reinterpretCast.....	36
6.1.8 staticCast.....	36
6.2 Component Context.....	36
6.2.1 getCurrent.....	37
6.2.2 getURI	37
6.2.3 getService.....	37
6.2.4 getServices.....	37
6.2.5 getServiceReference.....	38
6.2.6 getServiceReferences	38
6.2.7 getProperties	38
6.2.8 getDataFactory.....	39
6.2.9 getSelfReference.....	39
6.3 ServiceReference	39
6.3.1 getService.....	40
6.3.2 getCallback.....	40
6.4 DomainContext	40
6.4.1 getService.....	40
6.5 SCAException.....	41
6.5.1 getEClassName	41
6.5.2 getMessageText.....	41
6.5.3 getFileName	41
6.5.4 getLineNumber	42
6.5.5 getFunctionName	42
6.6 SCANullPointerException	42
6.7 ServiceRuntimeException.....	42
6.8 ServiceUnavailableException	43
6.9 MultipleServicesException.....	43
7 C++ Contributions.....	44
7.1 Executable files.....	44
7.1.1 Executable in contribution	44
7.1.2 Executable shared with other contribution(s) (Export)	44
7.1.3 Executable outside of contribution (Import).....	45
7.2 componentType files	46
7.3 C++ Contribution Extensions	47
7.3.1 Export.cpp	47
7.3.2 Import.cpp.....	47
8 C++ Interfaces	48
8.1 Types Supported in Service Interfaces.....	48
8.1.1 Local Service	48
8.1.2 Remotable Service	48
8.2 Header Files.....	48
9 WSDL to C++ and C++ to WSDL Mapping	49

9.1 Augmentations for WSDL to C++ Mapping	49
9.1.1 Mapping WSDL targetNamespace to a C++ namespace	49
9.1.2 Mapping WSDL Faults to C++ Exceptions	50
9.1.3 Mapping of in, out, in/out parts to C++ member function parameters.....	50
9.2 Augmentations for C++ to WSDL Mapping	50
9.2.1 Mapping C++ namespaces to WSDL namespaces	51
9.2.2 Parameter and return type classification.....	51
9.2.3 C++ to WSDL Type Conversion	51
9.2.4 Service-specific Exceptions.....	51
9.3 SDO Data Binding	51
9.3.1 Simple Content Binding	51
9.3.2 Complex Content Binding.....	53
10 Conformance	54
10.1 Conformance Targets	54
10.2 SCA Implementations	54
10.3 SCA Documents	55
10.4 C++ Files.....	55
10.5 WSDL Files	55
A C++ SCA Annotations	56
A.1 Application of Annotations to C++ Program Elements	56
A.2 Interface Header Annotations.....	56
A.2.1 @Interface	57
A.2.2 @Remotable	57
A.2.3 @Callback.....	57
A.2.4 @OneWay	58
A.2.5 @Function.....	59
A.3 Implementation Header Annotations	59
A.3.1 @ComponentType.....	59
A.3.2 @Scope	60
A.3.3 @EagerInit	60
A.3.4 @AllowsPassByReference	61
A.3.5 @Property	61
A.3.6 @Reference	62
A.4 Base Annotation Grammar.....	63
B C++ SCA Policy Annotations.....	64
B.1 General Intent Annotations.....	64
B.2 Specific Intent Annotations	65
B.2.1 Security Interaction	66
B.2.2 Security Implementation	66
B.2.3 Reliable Messaging.....	67
B.2.4 Transactions.....	67
B.2.5 Miscellaneous	67
B.3 Policy Set Annotations	67
B.4 Policy Annotation Grammar Additions	68
B.5 Annotation Constants	68

C	C++ WSDL Mapping Annotations	69
C.1	Interface Header Annotations	69
C.1.1	@WebService	69
C.1.2	@WebFunction	70
C.1.3	@OneWay	72
C.1.4	@WebParam	73
C.1.5	@WebResult	75
C.1.6	@SOAPBinding	77
C.1.7	@WebFault	78
C.1.8	@WebThrows	80
D	WSDL C++ Mapping Extensions	81
D.1	<cpp:bindings>	81
D.2	<cpp:class>	81
D.3	<cpp:enableWrapperStyle>	82
D.4	<cpp:namespace>	83
D.5	<cpp:memberFunction>	84
D.6	<cpp:parameter>	85
D.7	JAX-WS WSDL Extensions	87
D.8	sca-wsdl-ext-cpp-1.1.xsd	87
E	XML Schemas	89
E.1	sca-interface-cpp-1.1.xsd	89
E.2	sca-implementation-cpp-1.1.xsd	89
E.3	sca-contribution-cpp-1.1.xsd	90
F	Normative Statement Summary	92
F.1	Annotation Normative Statement Summary	95
F.2	WSDL Extention Normative Statement Summary	96
F.3	JAX-WS Normative Statements	96
F.3.1	Ignored Normative Statements	99
G	Migration	101
G.1	Method child elements of interface.cpp and implementation.cpp	101
H	Acknowledgements	102
I	Revision History	103

1 Introduction

2 This document describes the SCA Client and Implementation Model for the C++ programming language.
3 The SCA C++ implementation model describes how to implement SCA components in C++. A component
4 implementation itself can also be a client to other services provided by other components or external
5 services. The document describes how a C++ implemented component gets access to services and calls
6 their operations.
7 The document also explains how non-SCA C++ components can be clients to services provided by other
8 components or external services. The document shows how those non-SCA C++ component
9 implementations access services and call their operations.

10 1.1 Terminology

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

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

16

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/200912 "	Defined by the SCA specifications
cpp	" http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901 "	

17 Table 1-1: Prefixes and Namespaces used in this Specification

18 1.2 Normative References

- 19 [[RFC2119](#)] S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, IETF
20 RFC 2119, March 1997. <http://www.ietf.org/rfc/rfc2119.txt>
- 21 [[ASSEMBLY](#)] OASIS Committee Draft 05, *Service Component Architecture Assembly Model
22 Specification Version 1.1*, January 2010. <http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd05.pdf>
- 23 [[POLICY](#)] OASIS Committee Draft 02, *SCA Policy Framework Version 1.1*, March 2009.
24 <http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd02.pdf>
- 25 [[SDO21](#)] OSOA, *Service Data Objects For C++ Specification*, December 2006.
26 <http://www.osoa.org/download/attachments/36/CPP-SDO-Spec-v2.1.0-FINAL.pdf>
- 27 [[WSDL11](#)] World Wide Web Consortium, *Web Service Description Language (WSDL)*,
28 March 2001. <http://www.w3.org/TR/wsdl>
- 29 [[XSD](#)] World Wide Web Consortium, *XML Schema Part 2: Datatypes Second Edition*,
30 October 2004. <http://www.w3.org/TR/xmlschema-2/>
- 31 [[JAXWS21](#)] Doug. Kohlert and Arun Gupta, *The Java API for XML-Based Web Services
32 (JAX-WS) 2.1*, JSR, JCP, May 2007.
33 <http://jcp.org/aboutJava/communityprocess/mrel/jsr224/index2.html>

35 **1.3 Conventions**

36 **1.3.1 Naming Conventions**

37 This specification follows naming conventions for artifacts defined by the specification:

- 38 • For the names of elements and the names of attributes within XSD files, the names follow the
39 CamelCase convention, with all names starting with a lower case letter.
40 e.g. <element name="componentType" type="sca:ComponentType"/>
 - 41 • For the names of types within XSD files, the names follow the CamelCase convention with all names
42 starting with an upper case letter
43 e.g. <complexType name="ComponentService">
 - 44 • For the names of intents, the names follow the CamelCase convention, with all names starting with a
45 lower case letter, EXCEPT for cases where the intent represents an established acronym, in which
46 case the entire name is in upper case.
- 47 An example of an intent which is an acronym is the "SOAP" intent.

48 **1.3.2 Typographic Conventions**

49 This specification follows typographic conventions for specific constructs:

- 50 • Conformance points are highlighted, [numbered] and cross-referenced to Normative Statement
51 Summary
- 52 • XML attributes are identified in text as @attribute
- 53 • Language identifiers used in text are in courier
- 54 • Literals in text are in *italics*

55 2 Basic Component Implementation Model

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

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

61 2.1 Implementing a Service

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

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

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

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

70 Snippet 2-1 – Snippet 2-3 show a C++ service interface and the C++ implementation class of a C++
71 implementation.

```
72
73     // LoanService interface
74     class LoanService {
75     public:
76         virtual bool approveLoan(unsigned long customerNumber,
77                               unsigned long loanAmount) = 0;
78     };
```

79 *Snippet 2-1: A C++ Service Interface*

```
80
81     class LoanServiceImpl : public LoanService {
82     public:
83         LoanServiceImpl();
84         virtual ~LoanServiceImpl();
85
86         virtual bool approveLoan(unsigned long customerNumber,
87                               unsigned long loanAmount);
88     };
```

89 *Snippet 2-2: C++ Service Implementation Declaration*

```
90
91     #include "LoanServiceImpl.h"
92
93     LoanServiceImpl::LoanServiceImpl()
94     {
95         ...
96     }
97
98     LoanServiceImpl::~LoanServiceImpl()
99     {
100        ...
101    }
102
103    bool LoanServiceImpl::approveLoan(unsigned long customerNumber,
```

```

104             {
105                 ...
106             }
107         }
```

108 *Snippet 2-3: C++ Service Implementation*

109

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

111

```

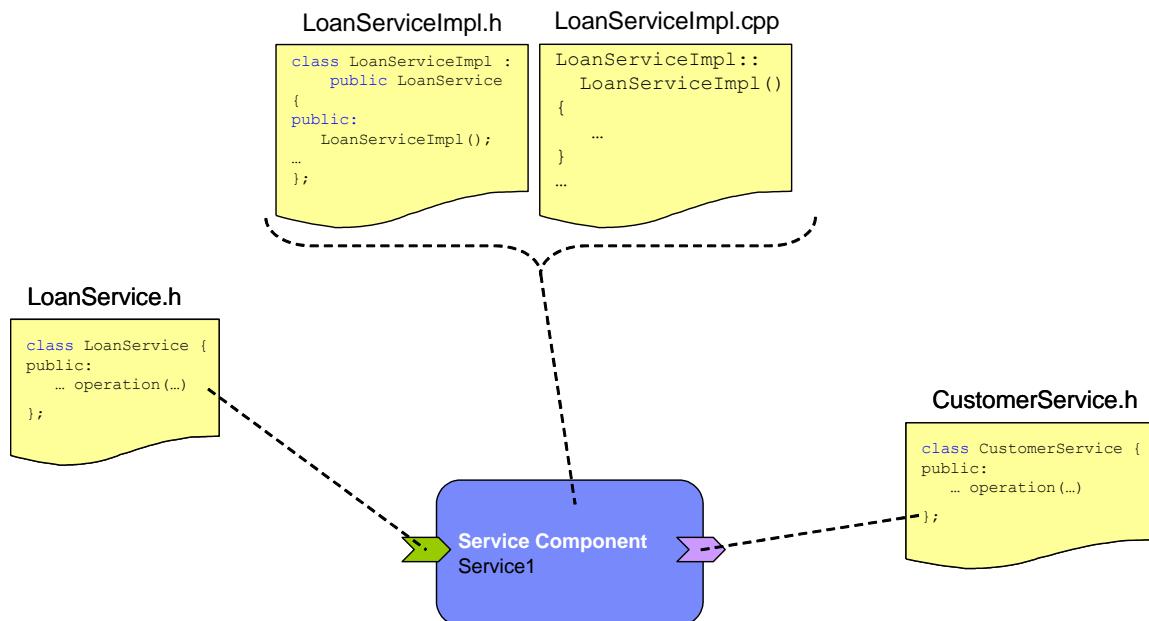
112 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912">
113     <service name="LoanService">
114         <interface.cpp header="LoanService.h"/>
115     </service>
116 </componentType>
```

117 *Snippet 2-4: Component Type for Service Implementation in Snippet 2-3*

118

119 Figure 2-1 shows the relationship between the C++ header files and implementation files for a component
120 that has a single service and a single reference.

121



122

123 *Figure 2-1: Relationship of C++ Implementation Artifacts*

2.1.1 Implementing a Remotable Service

125 A `@remotable="true"` attribute on an `interface.cpp` element indicates that the interface is **remotable** as
126 described in the Assembly Specification **[ASSEMBLY]**. Snippet 2-5 shows the component type for a
127 remotable service:

128

```

129 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912">
130     <service name="LoanService">
131         <interface.cpp header="LoanService.h" remotable="true"/>
132     </service>
133 </componentType>
```

134 *Snippet 2-5: ComponentType for a Remotable Service*

135 **2.1.2 AllowsPassByReference**

136 Calls to remotable services have by-value semantics. This means that input parameters passed to the
137 service can be modified by the service without these modifications being visible to the client. Similarly, the
138 return value or exception from the service can be modified by the client without these modifications being
139 visible to the service implementation. For remote calls (either cross-machine or cross-process), these
140 semantics are a consequence of marshalling input parameters, return values and exceptions “on the wire”
141 and unmarshalling them “off the wire” which results in physical copies being made. For local calls within
142 the same operating system address space, C++ calling semantics include by-reference and therefore do
143 not provide the correct by-value semantics for SCA remotable interfaces. To compensate for this, the
144 SCA runtime can intervene in these calls to provide by-value semantics by making copies of any by-
145 reference values passed.

146 The cost of such copying can be very high relative to the cost of making a local call, especially if the data
147 being passed is large. Also, in many cases this copying is not needed if the implementation observes
148 certain conventions for how input parameters, return values and exceptions are used. An
149 `@allowsPassByReference="true"` attribute allows implementations to indicate that they use input
150 parameters, return values and exceptions in a manner that allows the SCA runtime to avoid the cost of
151 copying by-reference values when a remotable service is called locally within the same operating system
152 address space. See Implementation.cpp and Implementation Function for a description of the
153 `@allowsPassByReference` attribute and how it is used.

154 **2.1.2.1 Marking services and references as “allows pass by reference”**

155 Marking a service member function implementation as “allows pass by reference” asserts that the
156 member function implementation observes the following restrictions:

- 157 • Member function execution will not modify any input parameter before the member function returns.
- 158 • The service implementation will not retain a reference or pointer to any by-reference input parameter,
159 return value or exception after the member function returns.
- 160 • The member function will observe “allows pass by value” client semantics, as defined in the next
161 paragraph for any callbacks that it makes.

162 Marking a client as “allows pass by reference” asserts that for all reference’s member functions, the client
163 observes the restrictions:

- 164 • The client implementation will not modify any member function’s input parameters before the member
165 function returns. Such modifications might occur in callbacks or separate client threads.
- 166 • If a member function is one-way, the client implementation will not modify any of the member
167 function’s input parameters at any time after calling the operation. This is because one-way member
168 function calls return immediately without waiting for the service member function to complete.

169 **2.1.2.2 Using “allows pass by reference” to optimize remotable calls**

170 The SCA runtime MAY use by-reference semantics when passing input parameters, return values or
171 exceptions on calls to remotable services within the same system address space if both the service
172 member function implementation and the client are marked “allows pass by reference”. [CPP20014]

173 The SCA runtime MUST use by-value semantics when passing input parameters, return values and
174 exceptions on calls to remotable services within the same system address space if the service member
175 function implementation is not marked “allows pass by reference” or the client is not marked “allows pass
176 by reference”. [CPP20015]

177 **2.1.3 Implementing a Local Service**

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

179 2.2 Component Implementation Scopes

180 SCA defines the concept of implementation scope, which specifies the lifecycle contract an
181 implementation has with the runtime. Invocations on a service offered by a component will be dispatched
182 by the SCA runtime to an implementation instance according to the semantics of its scope.

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

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

186 An SCA runtime MUST support these scopes; **stateless** and **composite**. Additional scopes MAY be
187 provided by SCA runtimes. [CPP2003]

188 Snippet 2-6 shows the component type for a composite scoped component:

```
190 <component name="LoanService">
191   <implementation.cpp library="loan" class="LoanServiceImpl"
192     scope="composite"/>
193 </component>
```

194 *Snippet 2-6: Component Type for a Composite Scoped Component*

195 Independent of scope, component implementations have to manage any state maintained in global
196 variables or static data members. A library can be loaded as early as when any component implemented
197 by the library enters the running state **[ASSEMBLY]** but no later than the first member function invocation
198 of a service provided by a component implemented by the library. Component implementations can not
199 make any assumptions about when a library might be unloaded. An SCA runtime MUST NOT perform
200 any synchronization of access to component implementations. [CPP20018]

202 2.2.1 Stateless Scope

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

205 The concurrency model for the stateless scope is single threaded. An SCA runtime MUST ensure that a
206 stateless scoped implementation instance object is only ever dispatched on one thread at any one time.
207 In addition, within the SCA lifecycle of an instance, an SCA runtime MUST only make a single invocation
208 of one business member function. [CPP20012]

209 2.2.2 Composite Scope

210 All service requests are dispatched to the same implementation instance for the lifetime of the composite
211 containing the component. The lifetime of the containing composite is defined as the time it placed in the
212 running state to the time it is removed from the running state, either normally or abnormally.

213 A composite scoped implementation can also specify eager initialization using the @eagerInit="true"
214 attribute on the *implementation.cpp* element of a component definition. When marked for eager
215 initialization, the implementation instance will be created when the component is placed in running state,
216 otherwise, initialization is lazy and the instance will be created when the first client request is received.

217 The concurrency model for the composite scope is multi-threaded. An SCA runtime MAY run multiple
218 threads in a single composite scoped implementation instance object. [CPP20013]

219 2.3 Implementing a Configuration Property

220 Component implementations can be configured through properties. The properties and their types (not
221 their values) are defined in the component type file. The C++ component can retrieve the properties using
222 the *getProperties()* on the *ComponentContext* class.

223 Snippet 2-7 shows how to get the property values.

224

```

225 #include "ComponentContext.h"
226 using namespace oasis::sca;
227
228 void clientFunction()
229 {
230     ...
231
232     ComponentContextPtr context = ComponentContext::getCurrent();
233
234     DataObjectPtr properties = context->getProperties();
235
236     long loanRating = properties->getInteger("maxLoanValue");
237
238     ...
239 }
```

240 *Snippet 2-7: Retrieving Property Values*

241 2.4 Component Type and Component

242 For a C++ component implementation, a component type is specified in a side file. By default, the
 243 componentType side file is in the root directory of the composite containing the component or some
 244 subdirectory of the composite root directory with a name matching the implementation class of the
 245 component. The location can be modified as described in Implementation.cpp.

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

248 Snippet 2-8 – Snippet 2-10 show a C++ service interface and the C++ implementation class of a C++
 249 service.

```

250
251 // LoanService interface
252 class LoanService {
253 public:
254     virtual bool approveLoan(unsigned long customerNumber,
255                             unsigned long loanAmount) = 0;
256 };
```

257 *Snippet 2-8: A C++ Service Interface*

```

258
259 class LoanServiceImpl : public LoanService {
260 public:
261     LoanServiceImpl();
262     virtual ~LoanServiceImpl();
263
264     virtual bool approveLoan(unsigned long customerNumber,
265                             unsigned long loanAmount);
266 };
```

267 *Snippet 2-9: C++ Service Implementation Declaration*

```

268
269 #include "LoanServiceImpl.h"
270
271 // Construction/Destruction
272
273 LoanServiceImpl::LoanServiceImpl()
274 {
275     ...
276 }
277 LoanServiceImpl::~LoanServiceImpl()
278 {
279     ...
```

```
280     }
281     // Implementation
282
283     bool LoanServiceImpl::approveLoan(unsigned long customerNumber,
284                                         unsigned long loanAmount)
285     {
286         ...
287     }
```

288 *Snippet 2-10: C++ Service Implementation*

289

290 Snippet 2-11 shows the component type for this component implementation.

291

```
292 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912">
293     <service name="LoanService">
294         <interface.cpp header="LoanService.h"/>
295     </service>
296 </componentType>
```

297 *Snippet 2-11: Component Type for Service Implementation in Snippet 2-10*

298

299 Snippet 2-12 shows a component using the implementation.

300

```
301 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
302     name="LoanComposite" >
303     ...
304
305     <component name="LoanService">
306         <implementation.cpp library="loan" class="LoanServiceImpl"/>
307     </component>
308 </composite>
```

309 *Snippet 2-12: Component Using Implementation in Snippet 2-10*

310 **2.4.1 Interface.cpp**

311 Snippet 2-13 shows the pseudo-schema for the C++ interface element used to type services and
312 references of component types.

313

```
314 <!-- interface.cpp schema snippet -->
315 <interface.cpp xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
316     header="string" class="Name"? remotable="boolean"?
317     callbackHeader="string" callbackClass="Name"?
318     requires="listOfQNames"? policySets="listOfQNames"? >
319
320     <function ... />*
321     <callbackFunction ... />*
322     <requires/>*
323     <policySetAttachment/>*
324
325 </interface.cpp>
```

326 *Snippet 2-13: Pseudo-schema for C++ Interface Element*

327

328 The **interface.cpp** element has the **attributes**:

- **header : string (1..1)** – full name of the header file that describes the interface, including relative path from the composite root.

- **class : Name (0..1)** – name of the class declaration for the interface in the header file, including any namespace definition. If the header file identified by the @header attribute of an *<interface.cpp/>* element contains more than one class, then the @class attribute MUST be specified for the *<interface.cpp/>* element. [CPP20005]
- **callbackHeader : string (0..1)** – full name of the header file that describes the callback interface, including relative path from the composite root.
- **callbackClass : Name (0..1)** – name of the class declaration for the callback interface in the callback header file, including any namespace definition. If the header file identified by the @callbackHeader attribute of an *<interface.cpp/>* element contains more than one class, then the @callbackClass attribute MUST be specified for the *<interface.cpp/>* element. [CPP20006]
- **remotable : boolean (0..1)** – indicates whether the service is remotable or local. The default is local. See Implementing a Remotable Service
- **requires : listOfQNames (0..1)** – a list of policy intents. See the Policy Framework specification [POLICY] for a description of this attribute. If intents are specified at both the class and member function level, the effective intents for the member function is determined by merging the combined intents from the member function with the combined intents for the class according to the Policy Framework rules for merging intents within a structural hierarchy, with the member function at the lower level and the class at the higher level.
- **policySets : listOfQNames (0..1)** – a list of policy sets. See the Policy Framework specification [POLICY] for a description of this attribute.

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

- **function : CPPFunction (0..n)** – see Function and CallbackFunction
- **callbackFunction : CPPFunction (0..n)** – see Function and CallbackFunction
- **requires : requires (0..n)** - See the Policy Framework specification [POLICY] for a description of this element.
- **policySetAttachment : policySetAttachment (0..n)** - See the Policy Framework specification [POLICY] for a description of this element.

2.4.2 Function and CallbackFunction

A member function of an interface might have behavioral characteristics that need to be identified. This is done using a *function* or *callbackFunction* child element of *interface.cpp*. These child elements are also used when not all functions in a class are part of the interface.

- If the header file identified by the @header attribute of an *<interface.cpp/>* element contains function declarations that are not operations of the interface, then the functions that are not operations of the interface MUST be excluded using *<function/>* child elements of the *<interface.cpp/>* element with @exclude="true". [CPP20016]
- If the header file identified by the @callbackHeader attribute of an *<interface.cpp/>* element contains function declarations that are not operations of the callback interface, then the functions that are not operations of the callback interface MUST be excluded using *<callbackFunction/>* child elements of the *<interface.cpp/>* element with @exclude="true". [CPP20017]

Snippet 2-14 shows the *interface.cpp* pseudo-schema with the pseudo-schema for the *function* and *callbackFunction* child elements:

```

<!-- Function schema snippet -->
<interface.cpp xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912" ... >

    <function name="NCName" requires="listOfQNames"? policySets="listOfQNames"?
        oneWay="Boolean"? exclude="Boolean"? >
        <requires/>*

```

```

380     <policySetAttachment/>*
381   </function> *
382
383   <callbackFunction name="NCName" requires="listOfQNames"?
384     policySets="listOfQNames"? oneWay="Boolean"? exclude="Boolean"? ?
385     <requires/>*
386     <policySetAttachment/>*
387   </callbackFunction> *
388
389 </interface.cpp>

```

390 Snippet 2-14: Pseudo-schema for Interface Function and CallbackFunction Sub-elements

- 391
- 392 The **function** and **callbackFunction** elements have the **attributes**:
- **name : NCName (1..1)** – name of the method being decorated. The @name attribute of a <function> child element of a <interface.cpp/> MUST be unique amongst the <function> elements of that <interface.cpp/>. [CPP20007]
- 393 The @name attribute of a <callbackFunction> child element of a <interface.cpp/> MUST be unique amongst the <callbackFunction> elements of that <interface.cpp/>. [CPP20008]
- **requires : listOfQNames (0..1)** – a list of policy intents. See the Policy Framework specification [POLICY] for a description of this attribute.
 - **policySets : listOfQNames (0..1)** – a list of policy sets. See the Policy Framework specification [POLICY] for a description of this attribute.
 - **oneWay : boolean (0..1)** – see Non-blocking Calls
 - **exclude : boolean (0..1)** – if true, the member function is excluded from the interface. The default is false.
- 405 The **function** and **callbackFunction** elements have the **child elements**:
- **requires : requires (0..n)** - See the Policy Framework specification [POLICY] for a description of this element.
 - **policySetAttachment : policySetAttachment (0..n)** - See the Policy Framework specification [POLICY] for a description of this element.

410 2.4.3 Implementation.cpp

411 Snippet 2-15 shows the pseudo-schema for the C++ implementation element used to define the
412 implementation of a component.

```

413
414 <!-- implementation.cpp schema snippet -->
415 <implementation.cpp xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
416   library="NCName" path="string"? class="Name"
417   scope="scope"? componentType="string"? allowsPassByReference="Boolean"?
418   eagerInit="boolean"? requires="listOfQNames"?
419   policySets="listOfQNames"? >
420
421   <function ... />*
422   <requires/>*
423   <policySetAttachment/>*
424
425 </implementation.cpp>

```

426 Snippet 2-15: Pseudo-schema for C++ Implementation Element

- 427
- 428 The **implementation.cpp** element has the **attributes**:

- 429 • **library : NCName (1..1)** – name of the dll or shared library that holds the factory for the service
 430 component. This is the root name of the library.
- 431 • **path : string (0..1)** - path to the library which is either relative to the root of the contribution containing
 432 the composite or is prefixed with a contribution import name and is relative to the root of the import.
 433 See C++ Contributions.
- 434 • **class : Name (1..1)** – name of the class declaration of the implementation, including any namespace
 435 definition. The name of the componentType file for a C++ implementation MUST match the class
 436 name (excluding any namespace definition) of the implementations as defined by the @class attribute
 437 of the `<implementation.cpp/>` element. [CPP2009] The SCA runtime will append .componentType to
 438 the class name to find the componentType file.
- 439 • **scope : CPPImplementationScope (0..1)** – identifies the scope of the component implementation.
 440 The default is stateless. See Component Implementation Scopes
- 441 • **componentType : string (0..1)** – path to the componentType file which is relative to the root of the
 442 contribution containing the composite or is prefixed with a contribution import name and is relative to
 443 the root of the import.
- 444 • **allowsPassByReference : boolean (0..1)** – indicates the implementation allows pass by reference
 445 data exchange semantics on calls to it or from it. These semantics apply to all services provided by
 446 and references used by an implementation. See AllowsPassByReference
- 447 • **eagerInit : boolean (0..1)** – indicates a composite scoped implementation is to be initialized when it
 448 is loaded. See Composite Scope
- 449 • **requires : listOfQNames (0..1)** – a list of policy intents. See the Policy Framework specification
 450 [POLICY] for a description of this attribute. If intents are specified at both the class and member
 451 function level, the effective intents for the member function is determined by merging the combined
 452 intents from the member function with the combined intents for the class according to the Policy
 453 Framework rules for merging intents within a structural hierarchy, with the member function at the
 454 lower level and the class at the higher level.
- 455 • **policySets : listOfQNames (0..1)** – a list of policy sets. See the Policy Framework specification
 456 [POLICY] for a description of this attribute.
- 457 The `implementation.cpp` element has the **child elements**:
- 458 • **function : CPPImplementationMethod (0..n)** – see Implementation Function
- 459 • **requires : requires (0..n)** - See the Policy Framework specification [POLICY] for a description of this
 460 element.
- 461 • **policySetAttachment : policySetAttachment (0..n)** - See the Policy Framework specification
 462 [POLICY] for a description of this element.

463 2.4.4 Implementation Function

464 A member function of an implementation might have operational characteristics that need to be identified.
 465 This is done using a `function` child element of `implementation.cpp`

466 Snippet 2-16 shows the `implementation.cpp` schema with the schema for a `method` child element:

467

```

468 <!-- ImplementationFunction schema snippet -->
469 <implementation.cpp xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912" ...
470 >
471
472     <function name="NCName" requires="listOfQNames"? policySets="listOfQNames"?
473         allowsPassByReference="boolean"? >
474         <requires/>*
475         <policySetAttachment/>*
476     </function> *
477
478 </implementation.cpp>

```

479 Snippet 2-16: Pseudo-schema for Implementation Function Sub-element

480

481 The **function** element has the **attributes**:

- 482 • **name : NCName (1..1)** – name of the method being decorated. The @name attribute of a
483 <function/> child element of a <implementation.cpp/> MUST be unique amongst the <function/>
484 elements of that <implementation.cpp/>. [CPP20010]
- 485 • **requires : listOfQNames (0..1)** – a list of policy intents. See the Policy Framework specification
486 [POLICY] for a description of this attribute.
- 487 • **policySets : listOfQNames (0..1)** – a list of policy sets. See the Policy Framework specification
488 [POLICY] for a description of this attribute.
- 489 • **allowsPassByReference : boolean (0..1)** – indicates the member function allows pass by reference
490 data exchange semantics. See AllowsPassByReference

491 The **function** element has the **child elements**:

- 492 • **requires : requires (0..n)** - See the Policy Framework specification [POLICY] for a description of this
493 element.
- 494 • **policySetAttachment : policySetAttachment (0..n)** - See the Policy Framework specification
495 [POLICY] for a description of this element.

496 2.5 Instantiation

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

499 3 Basic Client Model

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

502 3.1 Accessing Services from Component Implementations

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

504 Snippet 3-1 shows the `ComponentContext` C++ class with its `getService()` member function.

505

```
506 namespace oasis {
507     namespace sca {
508
509         class ComponentContext {
510             public:
511                 static ComponentContextPtr getCurrent();
512                 virtual ServiceProxyPtr getService(
513                     const std::string& referenceName) const = 0;
514
515             ...
516         }
517     }
```

518 *Snippet 3-1: Partial ComponetContext Class Definition*

519

520 The `getService()` member function takes as its input argument the name of the reference and returns
521 a pointer to a proxy providing access to the service. The returned pointer is to a generic `ServiceProxy`
522 and is assigned to a pointer to a proxy which is derived from `ServiceProxy` and implements the
523 interface of the reference.

524 Snippet 3-2 a sample of how the `ComponentContext` is used in a C++ component implementation. The
525 `getService()` member function is called on the `ComponentContext` passing the reference name as
526 input. The return of the `getService()` member function is cast to the abstract base class defined for the
527 reference.

528

```
529 #include "ComponentContext.h"
530 #include "CustomerServiceProxy.h"
531
532 using namespace oasis::sca;
533
534 void clientFunction()
535 {
536
537     unsigned long customerNumber = 1234;
538
539     ComponentContextPtr context = ComponentContext::getCurrent();
540
541     ServiceProxyPtr service = context->getService("customerService");
542     CustomerServiceProxyPtr customerService =
543         dynamicCast<CustomerServiceProxy>(service);
544
545     if (customerService)
546         short rating = customerService->getCreditRating(customerNumber);
547
548 }
```

549 Snippet 3-2: Using ComponentContext

550 3.2 Interface Proxies

551 For each Reference used by a client, a proxy class is generated by an SCA implementation. The proxy
552 class for a Reference is derived from both the base ServiceProxy and the interface of the Reference
553 (details below) and implements the necessary functionality to inform the SCA runtime that an operation is
554 being invoked and submit the request over the transport determined by the wiring.

555 The base ServiceProxy class definition (in the namespace oasis::sca) is:

556

```
557     class ServiceProxy {  
558         public:  
559             //Possible future extensions  
560     };
```

561 Snippet 3-3: ServiceProxy Class Definition

562

563 A remotable interface is always mappable to WSDL, which can be mapped to C++ as described in
564 WSDL to C++ and C++ to WSDL Mapping. The proxy class for a remotable interface is derived from
565 ServiceProxy and contains the member functions mapped from the WSDL definition for the interface. If
566 a remotable interface is defined with a C++ class, an SCA implementation SHOULD map the interface
567 definition to WSDL before generating the proxy for the interface. [CPP30001]

568 For the interface definition:

569

```
570 <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"  
571     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"  
572     xmlns:tns="http://www.example.org/"  
573     xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"  
574     targetNamespace="http://www.example.org/">  
575  
576     <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"  
577         xmlns:tns="http://www.example.org/"  
578         attributeFormDefault="unqualified"  
579         elementFormDefault="unqualified"  
580         targetNamespace="http://www.example.org/">  
581     <xsd:element name="GetLastTradePrice" type="tns:GetLastTradePrice"/>  
582     <xsd:element name="GetLastTradePriceResponse"  
583         type="tns:GetLastTradePriceResponse"/>  
584     <xsd:complexType name="GetLastTradePrice">  
585         <xsd:sequence>  
586             <xsd:element name="tickerSymbol" type="xsd:string"/>  
587         </xsd:sequence>  
588     </xsd:complexType>  
589     <xsd:complexType name="GetLastTradePriceResponse">  
590         <xsd:sequence>  
591             <xsd:element name="return" type="xsd:float"/>  
592         </xsd:sequence>  
593     </xsd:complexType>  
594 </xsd:schema>  
595  
596     <message name="GetLastTradePrice">  
597         <part name="parameters" element="tns:GetLastTradePrice">  
598             </part>  
599     </message>  
600  
601     <message name="GetLastTradePriceResponse">  
602         <part name="parameters" element="tns:GetLastTradePriceResponse">  
603             </part>
```

```

604     </message>
605
606     <portType name="StockQuote">
607         <cpp:bindings>
608             <cpp:class name="StockQuoteService"/>
609         </cpp:bindings>
610         <operation name="GetLastTradePrice">
611             <cpp:bindings>
612                 <cpp:memberFunction name="getTradePrice"/>
613             </cpp:bindings>
614             <input name="GetLastTradePrice" message="tns:GetLastTradePrice">
615             </input>
616             <output name="GetLastTradePriceResponse"
617                 message="tns:GetLastTradePriceResponse">
618             </output>
619         </operation>
620     </portType>
621 </definitions>

```

622 *Snippet 3-4: Sample WSDL Interface*

623

624 The resulting abstract proxy class is:

625

```

626     // @WebService(name="StockQuote", targetNamespace="http://www.example.org/"
627     //           serviceName="StockQuoteService")
628     class StockQuoteServiceProxy: public ServiceProxy {
629
630         // @WebFunction(operationName="GetLastTradePrice",
631         //           action="urn:GetLastTradePrice")
632         float getTradePrice(const std::string& tickerSymbol);
633     };

```

634 *Snippet 3-5: Proxy Class for Interface in Snippet 3-4*

635

636 The proxy class for a local interface is derived from `ServiceProxy` and contains the member functions
637 of the C++ class defining the interface. For the interface definition:

638

```

639     // LoanService interface
640     class LoanService {
641     public:
642         virtual bool approveLoan(unsigned long customerNumber,
643                               unsigned long loanAmount) = 0;
644     };

```

645 *Snippet 3-6: Sample C++ Interface*

646

647 The resulting proxy class is:

648

```

649     class LoanServiceProxy : public ServiceProxy {
650     public:
651         virtual bool approveLoan(unsigned long customerNumber,
652                               unsigned long loanAmount) = 0;
653     };

```

654 *Snippet 3-7: Proxy Class for Interface in Snippet 3-6*

655

656 For each reference of a component, an SCA implementation MUST generate a service proxy derived
657 from `ServiceProxy` that contains the operations of the reference's interface definition. [CPP30002]

658 3.3 Accessing Services from non-SCA Component Implementations

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

661 Snippet 3-8 shows a sample of how the `DomainContext` is used in non-SCA C++ code.

662

```
663 #include "DomainContext.h"
664 #include "CustomerServiceProxy.h"
665
666 void externalFunction()
667 {
668
669     unsigned long customerNumber = 1234;
670
671     DomainContextPtr context = myImplGetDomain("http://example.com/mydomain");
672
673     ServiceProxyPtr service = context->getService("customerService");
674     CustomerServiceProxyPtr customerService =
675         dynamicCast<CustomerServiceProxy>(service);
676
677     if (customerService)
678         short rating = customerService->getCreditRating(customerNumber);
679
680 }
```

681 *Snippet 3-8: Using DomianContext*

682

683 No SCA metadata is specified for the client. E.g. no binding or policies are specified. Non-SCA clients
684 cannot call services that use callbacks.

685 The SCA infrastructure decides which binding is used OR extended form of serviceURI is used:

- `componentName/serviceName/bindingName`

687 The function `myImplGetDomain()` in Snippet 3-8 is an example of how a non-SCA client might get a
688 `DomainContextPtr` and is not a function defined by this specification. The specific mechanism for how
689 an SCA runtime implementation returns a `DomainContextPtr` is not defined by this specification.

690 3.4 Calling Service Operations

691 Accessing Services from Component Implementations and Accessing Services from non-SCA
692 Component Implementations show how to get access to a service. Once you have access to the service,
693 calling an operation of the service is like calling a member function of a C++ class via a `ServiceProxy`.

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

697 3.5 Long Running Request-Response Operations

698 The Assembly Specification [ASSEMBLY] allows service interfaces or individual operations to be marked
699 **long-running** using an `@requires="asyncInvocation"` intent, with the meaning that the operation(s) might
700 not complete in any specified time interval, even when the operations are request-response operations.

701 A client calling such an operation has to be prepared for any arbitrary delay between the time a request is
702 made and the time the response is received. To support this kind of operation three invocation styles are
703 available: asynchronous – the client provides a response handler, polling – the client will poll the SCA
704 runtime to determine if a response is available, and synchronous – the SCA runtime handles suspension
705 of the main thread, asynchronously receiving the response and resuming the main thread. The details of
706 each of these styles are provided in the following sections.

707 For a service operation with signature

```

708     <return type> <function name>(<parameters>);
709 the asynchronous invocation style includes a member function in the interface proxy class
710     <proxy_class>::<response_message_name>Response <function name>Async(
711                                         <in_parameters>);
712 Snippet 3-9: AsynchronousInvocation Member Function Format
713
714 where <response_message_name>Response is the response class for the operation as defined by
715 Response Class. The client uses this member function to issue a request through the SCA runtime. The
716 response is returned immediately, and can be used to access the response when it becomes available.
717
718 An SCA runtime MUST include an asynchronous invocation member function for every operation of a
719 reference interface with a @requires="asyncInvocation" intent applied either to the operation or the
720 reference as a whole. [CPP30003]
721 Snippet 3-10 shows a sample proxy class interface providing an asynchronous API.
722
723 using namespace oasis::sca;
724 using namespace commonj::sdo;
725
726
727 class CustomerServiceProxy : public ServiceProxy {
728 public:
729
730     // synchronous member function
731     virtual short getCreditRating(unsigned long customerNumber) = 0;
732
733     // forward declare callback class
734     class getCreditRatingCallback;
735
736     // asynchronous response object
737     class getCreditRatingResponse {
738 public:
739         // IOU/Future member functions
740         virtual void cancel() = 0;
741         virtual bool isCancelled() = 0;
742         virtual bool isReady() = 0;
743         virtual void setCallback(getCreditRatingCallbackPtr callback) = 0;
744
745         virtual short getReturn() = 0;
746     };
747
748     // asynchronous callback object
749     class getCreditRatingCallback {
750 public:
751         virtual void invoke(getCreditRatingResponsePtr) = 0;
752     };
753
754     // asynchronous member function
755     virtual getCreditRatingResponsePtr getCreditRatingAsync(unsigned long
756                                         customerNumber) = 0;
757 }
758 Snippet 3-10: Proxy with an Asynchronous API
759
760 Snippet 3-11 shows a sample of how the asynchronous invocation style is used in a C++ component
761 implementation.
762

```

```

763 #include "ComponentContext.h"
764 #include "CustomerServiceProxy.h"
765
766 using namespace oasis::sca;
767
768 void clientFunction()
769 {
770     ComponentContextPtr context = ComponentContext::getCurrent();
771
772     ServiceProxyPtr service = context->getService("customerService");
773     CustomerServiceProxyPtr customerService =
774         dynamicCast<CustomerServiceProxy>(service);
775
776     if (customerService) {
777         getCreditRatingResponsePtr response =
778             customerService->getCreditRatingAsync(1234);
779
780         // ...
781     }
782 }
783

```

784 *Snippet 3-11: Using an Asynchronous API*

785

786 Once a response object has been returned, the user can use the provided API in order to set a callback
787 object to be invoked when the response is ready, to poll the variable waiting for the response to become
788 available, or to block the current thread waiting for the response to become available.

789 3.5.1 Response Callback

790 If a callback is specified on a response object, that callback will be invoked when the response is received
791 by the runtime. Snippet 3-12 demonstrates creating a response object and setting it on a response
792 instance:

```

793
794 #include "ComponentContext.h"
795 #include "CustomerServiceProxy.h"
796
797 using namespace oasis::sca;
798
799 class CreditRatingCallback :
800     public CustomerServiceProxy::getCreditRatingCallback {
801
802     virtual void invoke(getCreditRatingResponsePtr response) {
803         try {
804             short rating = response->getReturn();
805         }
806         catch (...) {
807             // ...
808         }
809     }
810 };
811
812 void clientFunction()
813 {
814     ComponentContextPtr context = ComponentContext::getCurrent();
815
816     ServiceProxyPtr service = context->getService("customerService");
817     CustomerServiceProxyPtr customerService =
818         dynamicCast<CustomerServiceProxy>(service);
819
820     if (customerService) {
821         CustomerServiceProxy::getCreditRatingResponsePtr response =

```

```

822         customerService->getCreditRatingAsync(1234);
823
824     CustomerServiceProxy::getCreditRatingCallbackPtr callback =
825         new CreditRatingCallback();
826
827     response->setCallback(callback);
828
829     // ...
830 }
831 }
```

832 *Snippet 3-12: Using a Response Object*

3.5.2 Response Polling

833 A client can poll a response object in order to determine if a response has been received.

```

835
836 #include "ComponentContext.h"
837 #include "CustomerServiceProxy.h"
838
839 using namespace oasis::sca;
840
841 void clientFunction()
842 {
843     ComponentContextPtr context = ComponentContext::getCurrent();
844
845     ServiceProxyPtr service = context->getService("customerService");
846     CustomerServiceProxyPtr customerService =
847         dynamicCast<CustomerServiceProxy>(service);
848
849     if (customerService) {
850         CustomerServiceProxy::getCreditRatingResponsePtr response =
851             customerService->getCreditRatingAsync(1234);
852
853         while (!response->isReady()) {
854             // do something else
855         }
856
857         // The response is ready and can be accessed without blocking.
858         try {
859             short rating = response->getReturn();
860         }
861         catch (...) {
862             // ...
863         }
864     }
865 }
```

866 *Snippet 3-13: Polling a Response Object*

3.5.3 Synchronous Response Access

867 If a client chooses to block until a response becomes available, they can attempt to access a part of the response object. If the response has not been received, the call will block until the response is available. Once the response is received and the response object is populated, the call will return.

```

871
872 #include "ComponentContext.h"
873 #include "CustomerServiceProxy.h"
874
875 using namespace oasis::sca;
876
877 void clientFunction()
```

```

878 {
879     ComponentContextPtr context = ComponentContext::getCurrent();
880
881     ServiceProxyPtr service = context->getService("customerService");
882     CustomerServiceProxyPtr customerService =
883         dynamicCast<CustomerServiceProxy>(service);
884
885     if (customerService) {
886         CustomerServiceProxy::getCreditRatingResponsePtr response =
887             customerService->getCreditRatingAsync(1234);
888
889         // The response is ready and can be accessed without blocking.
890         try {
891             short rating = response->getReturn();
892         }
893         catch (...) {
894             // ...
895         }
896     }
897 }
```

898 *Snippet 3-14: Blocking on a Response Object*

899 3.5.4 Response Class

900 The proxy for an interface includes a response class for a response message type returned by an
 901 operation that can be invoked asynchronously. A response class presents the following interface to the
 902 client component.

903

```

904 class <response_message_name>Response {
905 public:
906     virtual <response_message_type> getReturn() const = 0;
907     virtual void setCallback(<response_message_name>CallbackPtr callback) = 0;
908     virtual boolean isReady() const = 0;
909     virtual void cancel() const = 0;
910     virtual boolean isCancelled() const = 0;
911 };
```

912 *Snippet 3-15: AsynchronousInvocation Response Class Definition*

913

914 An SCA runtime MUST include a response class for every response message of a reference interface
 915 that can be returned by an operation of the interface with a `@requires="asyncInvocation"` intent applied
 916 either to the operation of the reference as a whole. [CPP30004]

917 3.5.4.1 `getReturn`

918 A C++ component implementation uses `getReturn()` to retrieve the response data for an asynchronous
 919 invocation.

Precondition	C++ component instance is running and has an outstanding asynchronous call	
Input Parameter		
Return	Response data for the operation.	
Throws	Any exceptions defined for the operation.	
Post Condition	The response object is marked as done.	

920 *Table 3-1: AsynchronousInvocation Response::getReturn Details*

921 **3.5.4.2 setCallback**

922 A C++ component implementation uses `setCallback()` to set a callback object for an asynchronous
923 invocation.

Precondition	C++ component instance is running and has an outstanding asynchronous call	
Input Parameter	callback	A pointer to the callback object for the response
Return		
Post Condition	The response object is marked as done.	

924 *Table 3-2: AsynchronousInvocation Response::setCallback Details*

925 **3.5.4.3 isReady**

926 A C++ component implementation uses `isReady()` to determine if the response data for an polling
927 invocation is available.

Precondition	C++ component instance is running and has an outstanding polling call	
Input Parameter		
Return	True if the response is avaialble	
Post Condition	No change	

928 *Table 3-3: AsynchronousInvocation Response::isReady Details*

929 **3.5.4.4 cancel**

930 A C++ component implementation uses `cancel()` to cancel an outstanding invocation.

Precondition	C++ component instance is running and has an outstanding asynchronous call	
Input Parameter		
Return		
Post Condition	If a response is subsequently received for the operation, it will be discarded.	

931 *Table 3-4: AsynchronousInvocation Response::cancel Details*

932 **3.5.4.5 isCancelled**

933 A C++ component implementation uses `isCancelled()` to determine if another thread has cancelled an
934 outstanding invocation.

Precondition	C++ component instance is running and has an outstanding asynchronous call	
Input Parameter		
Return	True if the operation has been cancelled	
Post Condition	No change	

935 *Table 3-5: AsynchronousInvocation Response::isCancelled Details*

936 4 Asynchronous Programming

937 Asynchronous programming of a service is where a client invokes a service and carries on executing
938 without waiting for the service to execute. Typically, the invoked service executes at some later time.
939 Output from the invoked service, if any, is fed back to the client through a separate mechanism, since no
940 output is available at the point where the service is invoked. This is in contrast to the call-and-return style
941 of synchronous programming, where the invoked service executes and returns any output to the client
942 before the client continues. The SCA asynchronous programming model consists of support for non-
943 blocking operation calls and callbacks.

944 4.1 Non-blocking Calls

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

948 Any member function that returns `void`, has only by-value parameters and has no declared exceptions
949 can be marked with the `@oneWay="true"` attribute in the interface definition of the service. An operation
950 marked as `oneWay` is considered non-blocking and the SCA runtime MAY use a binding that buffers the
951 requests to the member function and sends them at some time after they are made. [CPP40001]

952 Snippet 4-1 shows the component type for a service with the `reportEvent()` member function declared
953 as a one-way operation:

```
954
955 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912">
956     <service name="LoanService">
957         <interface.cpp header="LoanService.h">
958             <function name="reportEvent" oneWay="true" />
959         </interface.cpp>
960     </service>
961 </componentType>
```

962 *Snippet 4-1: ComponentType with oneWay Member Function*

963

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

968 4.2 Callbacks

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

971 A callback interface is declared by the `@callbackHeader` and `@callbackClass` attributes in the interface
972 definition of the service. Snippet 4-2 shows the component type for a service `MyService` with the interface
973 defined in `MyService.h` and the interface for callbacks defined in `MyServiceCallback.h`,

```
974
975 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912" >
976     <service name="MyService">
977         <interface.cpp header="MyService.h"
978             callbackHeader="MyServiceCallback.h"/>
979     </service>
980 </componentType>
```

981 *Snippet 4-2: ComponentType with a Callback Interface*

982 **4.2.1 Using Callbacks**

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

987 Snippet 4-3– Snippet 4-5 show a scenario in which bidirectional interfaces and callbacks could be used.
988 A client requests a quotation from a supplier. To process the enquiry and return the quotation, some
989 suppliers might need additional information from the client. The client does not know which additional
990 items of information will be needed by different suppliers. This interaction can be modeled as a
991 bidirectional interface with callback requests to obtain the additional information.

992

```
993     class Quotation {
994     public:
995         virtual double requestQuotation(std::string productCode,
996                                         unsigned int quantity) = 0;
997     };
998
999     class QuotationCallback {
1000    public:
1001        virtual std::string getState() = 0;
1002        virtual std::string getZipCode() = 0;
1003        virtual std::string getCreditRating() = 0;
1004    };

```

1005 *Snippet 4-3: C++ Interface with a Callback Interface*

1006

1007 In Snippet 4-3, the `requestQuotation` operation requests a quotation to supply a given quantity of a
1008 specified product. The `QuotationCallBack` interface provides a number of operations that the supplier can
1009 use to obtain additional information about the client making the request. For example, some suppliers
1010 might quote different prices based on the state or the zip code to which the order will be shipped, and
1011 some suppliers might quote a lower price if the ordering company has a good credit rating. Other
1012 suppliers might quote a standard price without requesting any additional information from the client.

1013 Snippet 4-4 illustrates a possible implementation of the example service.

1014

```
1015 #include "QuotationImpl.h"
1016 #include "QuotationCallbackProxy.h"
1017 #include "ComponentContext.h"
1018 using namespace oasis::sca;
1019
1020 double QuotationImpl::requestQuotation(std::string productCode,
1021                                         unsigned int quantity) {
1022     double price = getPrice(productQuote, quantity);
1023     double discount = 0;
1024
1025     ComponentContextPtr context = ComponentContext::getCurrent();
1026     ServiceReferencePtr serviceRef = context->getSelfReference();
1027     ServiceProxyPtr callback = serviceRef->getCallback();
1028     QuotationCallbackQuotationCallbackProxyPtr quotationCallback =
1029         dynamicCast<QuotationCallbackProxy>(callback);
1030
1031     if (quotationCallback) {
1032         if (quantity > 1000 && callback->getState().compare("FL") == 0)
1033             discount = 0.05;
1034         if (quantity > 10000 && callback->getCreditRating().data() == 'A')
1035             discount += 0.05;
1036     }
1037     return price * (1-discount);
1038 }
```

1039 Snippet 4-4: Implementation of Forward Service with Interface in Snippet 4-3

1040

1041 Snippet 4-5 is taken from the client of this example service. The client's service implementation class
1042 implements the member functions of the QuotationCallback interface as well as those of its own service
1043 interface ClientService.

1044

```
1045 #include "QuotationCallback.h"
1046 #include "QuotationServiceProxy.h"
1047 #include "ComponentContext.h"
1048 using namespace oasis::sca;
1049
1050 void ClientImpl:: aClientFunction() {
1051     ComponentContextPtr context = ComponentContext::getCurrent();
1052
1053     ServiceProxyPtr service = context->getService("quotationService");
1054     QuotationServiceProxyPtr quotationService =
1055         dynamicCast<QuotationServiceProxy>(service);
1056
1057     if (quotationService)
1058         quotationService->requestQuotation("AB123", 2000);
1059 }
1060
1061 std::string QuotationCallbackImpl::getState() {
1062     return "TX";
1063 }
1064 std::string QuotationCallbackImpl::getZipCode() {
1065     return "78746";
1066 }
1067 std::string QuotationCallbackImpl::getCreditRating() {
1068     return "AA";
1069 }
```

1070 Snippet 4-5: Implementation of Callback Interface in Snippet 4-3

1071

1072 For each service of a component that includes a bidirectional interface, an SCA implementation MUST
1073 generate a service proxy derived from `ServiceProxy` that contains the operations of the reference's
1074 callback interface definition. [CPP40002]

1075 If a service of a component that has a callback interface contains operations with a
1076 `@requires="asyncInvocation"` intent applied either to the operation of the reference as a whole, an SCA
1077 implementation MUST include asynchronous invocation member functions and response classes as
1078 described in Long Running Request-Response Operations. [CPP40003]

1079 In the example the callback is **stateless**, i.e., the callback requests do not need any information relating
1080 to the original service request. For a callback that needs information relating to the original service
1081 request (a **stateful** callback), this information can be passed to the client by the service provider as
1082 parameters on the callback request.

1083 4.2.2 Callback Instance Management

1084 Instance management for callback requests received by the client of the bidirectional service is handled in
1085 the same way as instance management for regular service requests. If the client implementation has
1086 stateless scope, the callback is dispatched using a newly initialized instance. If the client implementation
1087 has composite scope, the callback is dispatched using the same shared instance that is used to dispatch
1088 regular service requests.

1089 As described in Using Callbacks, a stateful callback can obtain information relating to the original service
1090 request from parameters on the callback request. Alternatively, a composite-scoped client could store
1091 information relating to the original request as instance data and retrieve it when the callback request is
1092 received. These approaches could be combined by using a key passed on the callback request (e.g., an

1093 order ID) to retrieve information that was stored in a composite-scoped instance by the client code that
1094 made the original request.

1095 **4.2.3 Implementing Multiple Bidirectional Interfaces**

1096 Since it is possible for a single class to implement multiple services, it is also possible for callbacks to be
1097 defined for each of the services that it implements. To access the callbacks the
1098 `ServiceReference::getCallback(serviceName)` member function is used, passing in the name
1099 of the service for which the callback is to be obtained.

5 Error Handling

Clients calling service operations will experience business exceptions, and SCA runtime exceptions. Business exceptions are raised by the implementation of the called service operation. It is expected that these will be caught by client invoking the operation on the service. SCA runtime exceptions are raised by the SCA runtime and signal problems in the management of the execution of components, and in the interaction with remote services. Currently the SCA runtime exceptions defined are:

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

6 C++ API

1121 All the C++ interfaces are found in the namespace `oasis::sca`, which has been omitted from the
1122 definitions for clarity.

6.1 Reference Counting Pointers

1125 These are a derived version of the familiar smart-pointer. The pointer class holds a real (dumb) pointer to
1126 the object. If the reference counting pointer is copied, then a duplicate pointer is returned with the same
1127 real pointer. A reference count within the object is incremented for each copy of the pointer, so only when
1128 all pointers go out of scope will the object be freed.

1129 Reference counting pointers in SCA have the same name as the type they are pointing to, with a suffix of
1130 `Ptr`. (E.g. `ComponentContextPtr`, `ServiceReferencePtr`).

1131 `RefCountingPointer` defines member functions with raw pointer like semantics. This includes
1132 defining operators for dereferencing the pointer (`*`, `->`), as well as operators for determining the validity of
1133 the pointer.

1134

```
1135 template <typename T>
1136 class RefCountingPointer {
1137 public:
1138     T& operator* () const;
1139     T* operator-> () const;
1140     operator void* () const;
1141     bool operator! () const;
1142 };
1143
1144 template <typename T, typename U>
1145 RefCountingPointer<T> constCast(RefCountingPointer<U> other);
1146
1147 template <typename T, typename U>
1148 RefCountingPointer<T> dynamicCast(RefCountingPointer<U> other);
1149
1150 template <typename T, typename U>
1151 RefCountingPointer<T> reinterpretCast(RefCountingPointer<U> other);
1152
1153 template <typename T, typename U>
1154 RefCountingPointer<T> staticCast(RefCountingPointer<U> other);
```

1155 *Snippet 6-1: RefCountingPointer Class Definition*

6.1.1 operator*

1157 A C++ component implementation uses the `*` operator to dereferences the underlying pointer of a
1158 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

1159 *Table 6-1: RefCountingPointer operator* Details*

1160 **6.1.2 operator->**

1161 A C++ component implementation uses the `->` operator to invoke member functions on the underlying
1162 pointer of a reference counting pointer. This is equivalent to invoking `p->func()` where `func()` is a
1163 member function defined on the underlying pointer type.

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

1164 *Table 6-2: RefCountingPointer operator-> Details*

1165 **6.1.3 operator void***

1166 A C++ component implementation uses the `void*` operator to determine if the underlying pointer of a
1167 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

1168 *Table 6-3: RefCountingPointer operator void* Details*

1169 **6.1.4 operator!**

1170 A C++ component implementation uses the `!` operator to determine if the underlying pointer of a
1171 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

1172 *Table 6-4: RefCountingPointer operator! Details*

1173 **6.1.5 constCast**

1174 The `constCast` global function provides the semantic behavior of the `const_cast` operator for use with
1175 `RefCountingPointers`.

Precondition	C++ component instance is running and has a reference counting pointer
Return	A <code>RefCountingPointer</code> instance templatized on the target type.
Post Condition	No change

1176 *Table 6-5: constCast for RefCountingPointers Details*

1177 **6.1.6 dynamicCast**

1178 The `dynamicCast` global function provides the semantic behavior of the `dynamic_cast` operator for use
1179 with `RefCountingPointers`.

Precondition	C++ component instance is running and has a reference counting pointer
--------------	--

Return	A RefCountingPointer instance templatized on the target type. This can return an invalid RefCountingPointer instance if the cast fails.
Post Condition	No change

1180 *Table 6-6: dynamicCast for RefCountingPointers Details*

6.1.7 reinterpretCast

1182 The reinterpretCast global function provides the semantic behavior of the reinterpret_cast operator for
1183 use with RefCountingPointers.

Precondition	C++ component instance is running and has a reference counting pointer
Return	A RefCountingPointer instance templatized on the target type.
Post Condition	No change

1184 *Table 6-7: reinterpretCast for RefCountingPointers Details*

6.1.8 staticCast

1186 The staticCast global function provides the semantic behavior of the static_cast operator for use with
1187 RefCountingPointers.

Precondition	C++ component instance is running and has a reference counting pointer
Return	A RefCountingPointer instance templatized on the target type.
Post Condition	No change

1188 *Table 6-8: staticCast for RefCountingPointers Details*

6.2 Component Context

1190 The complete ComponentContext interface definition is:

1191

```

1192     class ComponentContext {
1193     public:
1194         static ComponentContextPtr getCurrent();
1195
1196         virtual std::string getURI() const = 0;
1197
1198         virtual ServiceProxyPtr getService(
1199                         const std::string& referenceName) const = 0;
1200         virtual std::vector<ServiceProxyPtr> getServices(
1201                         const std::string& referenceName) const = 0;
1202
1203         virtual ServiceReferencePtr getServiceReference(
1204                         const std::string& referenceName) const = 0;
1205         virtual std::vector<ServiceReferencePtr> getServiceReferences(
1206                         const std::string& referenceName) const = 0;
1207
1208
1209         virtual DataObjectPtr getProperties() const = 0;
1210         virtual DataFactoryPtr getDataFactory() const = 0;
1211
1212         virtual ServiceReferencePtr getSelfReference() const = 0;
1213         virtual ServiceReferencePtr getSelfReference(
1214                         const std::string& serviceName) const = 0;
1215     };

```

1216 Snippet 6-2: *ComponentContext Class Definition*

6.2.1 getCurrent

1218 A C++ component implementation uses `ComponentContext::getCurrent()` to get a
1219 `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.	

1220 *Table 6-9: ComponentContext::getCurrent Details*

6.2.2 getURI

1222 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	

1223 *Table 6-10: ComponentContext::getURI Details*

6.2.3 getService

1225 A C++ component implementation uses `getService()` to get a service proxy implementing the interface
1226 defined for a Reference.

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

1227 *Table 6-11: ComponentContext::getService Details*

6.2.4 getServices

1229 A C++ component implementation uses `getServices()` to get a vector of service proxies implementing
1230 the interface defined for a Reference.

Precondition	C++ component instance is running and has a <code>ComponentContext</code>	
Input Parameter	referenceName	Name of the Reference to get an interface object for
Return	Vector of pointers to <code>ServiceProxy</code> objects implementing the interface of the	

	Reference. This vector will be empty if <code>referenceName</code> is not defined for the component. Operations need to be invoked on each object in the vector.
Post Condition	ServiceProxy objects for the Reference are constructed. These ServiceProxy objects are independent of any ServiceReferences that are obtained for the Reference.

1231 *Table 6-12: ComponentContext::getServices Details*

6.2.5 getServiceReference

1233 A C++ component implementation uses `getServiceReference()` to get a `ServiceReference` for a
1234 Reference.

Precondition	<code>C++ component instance is running and has a 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.	

1235 *Table 6-13: ComponentContext::getServiceReference Details*

6.2.6 getServiceReferences

1237 A C++ component implementation uses `getServiceReferences()` to get a vector of
1238 `ServiceReference` for a Reference.

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

1239 *Table 6-14: ComponentContext::getServiceReferences Details*

6.2.7 getProperties

1241 A C++ component implementation uses `getProperties()` to get its configured property values.

Precondition	<code>C++ component instance is running and has a 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.	

1242 *Table 6-15: ComponentContext::getProperties Details*

1243 **6.2.8 getDataFactory**

1244 A C++ component implementation uses `getDataFactory()` to get its an SDO `DataFactory` which
1245 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

1246 *Table 6-16: ComponentContext::getDataFactory Details*

1247 **6.2.9 getSelfReference**

1248 A C++ component implementation uses `getSelfReference()` to get a `ServiceReference` for use
1249 with some callback APIs.

1250

1251 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 Service
Post Condition	A <code>ServiceReference</code> object is constructed

1252 and

Precondition	C++ component instance is running and has a <code>ComponentContext</code>	
Input Parameter	<code>serviceName</code>	Name of the Service 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	

1253 *Table 6-17: ComponentContext::getSelfReference Details*

1254 **6.3 ServiceReference**

1255 The `ServiceReference` interface definition is:

1256

```
1257     class ServiceReference {
1258     public:
1259         virtual ServiceProxyPtr getService() const = 0;
1260
1261         virtual ServiceProxyPtr getCallback() const = 0;
1262     };
```

1263 *Snippet 6-3: ServiceReference Class Definition*

1264

1265 The detailed description of the usage of these member functions is described in Asynchronous
1266 Programming.

1267 **6.3.1 getService**

1268 A C++ component implementation uses `getService()` to get a service proxy implementing the interface
1269 defined for a `ServiceReference`.

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

1270 *Table 6-18: ServiceReference::getService Details*

1271 **6.3.2 getCallback**

1272 A C++ component implementation uses `getCallback()` to get a service proxy implementing the
1273 callback interface defined for a `ServiceReference`.

Precondition	C++ component instance is running and has a <code>ServiceReference</code>	
Input Parameter		
Return	Pointer to a <code>ServiceProxy</code> implementing the callback interface of the <code>ServiceReference</code> . This will be NULL if no callback interface is defined.	
Post Condition	A <code>ServiceProxy</code> object for the callback interface of the <code>ServiceReference</code> is constructed.	

1274 *Table 6-19: ServiceReference::getCallback Details*

1275 **6.4 DomainContext**

1276 The `DomainContext` interface definition is:

```
1277 class DomainContext {
1278 public:
1279     virtual ServiceProxyPtr getService(
1280             const std::string& serviceURI) const = 0;
1281 };
```

1282 *Snippet 6-4: DomainContext Class Definition*

1283 **6.4.1 getService**

1284 Non-SCA C++ code uses `getService()` to get a service proxy implementing the interface of a service
1285 in an SCA domain.

Precondition	None	
Input Parameter	serviceURI	URI of the Service to get an interface object for
Return	Pointer to a <code>ServiceProxy</code> object implementing the interface of the Service. This will be NULL if <code>serviceURI</code> is not defined in the domain.	

Post Condition	A ServiceProxy object for the Service is constructed.
----------------	---

1286 *Table 6-20: DomainContext::getService Details*

6.5 SCAException

1288 The SCAException interface definition is:

1289

```
1290     class SCAException : public std::exception {
1291     public:
1292         const char* getEClassName() const;
1293         const char* getMessageText() const;
1294         const char* getFileName() const;
1295         unsigned long getLineNumber() const;
1296         const char* getFunctionName() const;
1297     };
```

1298 *Snippet 6-5: SCAException Class Definition*

1299

1300 The details concerning this class and its derived types are described in Error Handling.

6.5.1 getEClassName

1301 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

1303 *Table 6-21: SCAException::getEClassName Details*

6.5.2 getMessageText

1304 A C++ component implementation uses `getMessageText()` to get any message included with the 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

1305 *Table 6-22: SCAException::getMessageText Details*

6.5.3 getFileName

1306 A C++ component implementation uses `getFileName()` to get the filename containing the function 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

1311 *Table 6-23: SCAException::getFileName Details*

6.5.4 getLineNumber

1313 A C++ component implementation uses `getLineNumber()` to get the line number in the source file
 1314 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

1315 *Table 6-24: SCAException::getLineNumber Details*

6.5.5 getFunctionName

1317 A C++ component implementation uses `getFunctionName()` to get the function name where the
 1318 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

1319 *Table 6-25: SCAException::getFunctionName Details*

6.6 SCANullPointerException

1321 The `SCANullPointerException` interface definition is:

1322

```
1323 class SCANullPointerException : public SCAException {  
1324 };
```

1325 *Snippet 6-6: SCANullPointerException Class Definition*

6.7 ServiceRuntimeException

1327 The `ServiceRuntimeException` interface definition is:

1328

```
1329 class ServiceRuntimeException : public SCAException {  
1330 };
```

1331 *Snippet 6-7: ServiceRuntimeException Class Definition*

1332 **6.8 ServiceUnavailableException**

1333 The ServiceUnavailableException interface definition is:

1334

```
1335     class ServiceUnavailableException : public ServiceRuntimeException {  
1336     };
```

1337 *Snippet 6-8: ServiceUnavailableException Class Definition*

1338 **6.9 MultipleServicesException**

1339 The MultipleServicesException interface definition is:

1340

```
1341     class MultipleServicesException : public ServiceRuntimeException {  
1342     };
```

1343 *Snippet 6-9: MultipleServicesException Class Definition*

1344 7 C++ Contributions

1345 Contributions are defined in the Assembly specification [ASSEMBLY]. C++ contributions are typically, but
1346 not necessarily contained in .zip files. In addition to SCDL and potentially WSDL artifacts, C++
1347 contributions include binary executable files, componentType files and potentially C++ interface headers.
1348 No additional discussion is needed for header files, but there are additional considerations for executable
1349 and componentType files.

1350 7.1 Executable files

1351 Executable files containing the C++ implementations for a contribution can be contained in the
1352 contribution, contained in another contribution or external to any contribution. In some cases, it could be
1353 desirable to have contributions share an executable. In other cases, an implementation deployment
1354 policy might dictate that executables are placed in specific directories in a file system.

1355 7.1.1 Executable in contribution

1356 When the executable file containing a C++ implementation is in the same contribution, the *@path*
1357 attribute of the *implementation.cpp* element is used to specify the location of the executable. The specific
1358 location of an executable within a contribution is not defined by this specification.

1359 Snippet 7-1 shows a contribution containing a DLL.

```
1360
1361     META-INF/
1362         sca-contribution.xml
1363     bin/
1364         autoinsurance.dll
1365     AutoInsurance/
1366         AutoInsurance.composite
1367         AutoInsuranceService/
1368             AutoInsurance.h
1369             AutoInsuranceImpl.componentType
1370         include/
1371             Customers.h
1372             Underwriting.h
1373             RateUtils.h
```

1374 *Snippet 7-1: Contribution Containing a DLL*

1375

1376 The SCDL for the AutoInsuranceService component of Snippet 7-1 is:

```
1377
1378     <component name="AutoInsuranceService">
1379         <implementation.cpp library="autoinsurance" path="bin/" 
1380             class="AutoInsuranceImpl" />
1381     </component>
```

1382 *Snippet 7-2: Component Definition Using Implementation in a Common DLL*

1383 7.1.2 Executable shared with other contribution(s) (Export)

1384 If a contribution contains an executable that also implements C++ components found in other
1385 contributions, the contribution has to export the executable. An executable in a contribution is made
1386 visible to other contributions by adding an **export.cpp** element to the contribution definition as shown in
1387 Snippet 7-3.

```
1388
1389     <contribution>
```

```
1390     <deployable composite="myNS:RateUtilities"
1391         <export.cpp name="contribNS:rates" >
1392     </contribution>
```

1393 *Snippet 7-3: Exporting a Contribution*

1394

1395 It is also possible to export only a subtree of a contribution. For a contribution:

1396

```
1397     META-INF/
1398         sca-contribution.xml
1399     bin/
1400         rates.dll
1401     RateUtilities/
1402         RateUtilities.composite
1403         RateUtilitiesService/
1404             RateUtils.h
1405             RateUtilsImpl.componentType
```

1406 *Snippet 7-4: Contribution with a Subdirectory to be Shared*

1407

1408 An export of the form:

1409

```
1410     <contribution>
1411         <deployable composite="myNS:RateUtilities"
1412             <export.cpp name="contribNS:ratesbin" path="bin/" >
1413     </contribution>
```

1414 *Snippet 7-5: Exporting a Subdirectory of a Contribution*

1415

1416 only makes the contents of the bin directory visible to other contributions. By placing all of the executable
1417 files of a contribution in a single directory and exporting only that directory, the amount of information
1418 available to a contribution that uses the exported executable files is limited. This is considered a best
1419 practice.

1420 7.1.3 Executable outside of contribution (Import)

1421 When the executable that implements a C++ component is located outside of a contribution, the
1422 contribution has to import the executable. If the executable is located in another contribution, the
1423 **import.cpp** element of the contribution definition uses a **@location** attribute that identifies the name of the
1424 export as defined in the contribution that defined the export as shown in Snippet 7-6.

1425

```
1426     <contribution>
1427         <deployable composite="myNS:Underwriting"
1428             <import.cpp name="rates" location="contribNS:rates">
1429     </contribution>
```

1430 *Snippet 7-6: Contribution with an Import*

1431

1432 The SCDL for the UnderwritingService component of Snippet 7-6 is:

1433

```
1434     <component name="UnderwritingService">
1435         <implementation.cpp library="rates" path="rates:bin/"
1436             class="UnderwritingImpl" />
1437     </component>
```

1438 *Snippet 7-7: Component Definition Using Implementation in an External DLL*

1439
1440 If the executable is located in the file system, the `@location` attribute identifies the location in the files
1441 system used as the root of the import as shown in Snippet 7-8.

1442
1443 <contribution>
1444 <deployable composite="myNS:CustomerUtilities"
1445 <import.cpp name="usr-bin" location="/usr/bin/" />
1446 </contribution>

1447 *Snippet 7-8: Component Definition Using Implementation in a File System*

1448 **7.2 componentType files**

1449 As stated in Component Type and Component, each component implemented in C++ has a
1450 corresponding componentType file. This componentType file is, by default, located in the root directory of
1451 the composite containing the component or a subdirectory of the composite root with the name
1452 `<implementation class>.componentType`, as shown in Snippet 7-9.

1453
1454 META-INF/
1455 sca-contribution.xml
1456 bin/
1457 autoinsurance.dll
1458 AutoInsurance/
1459 AutoInsurance.composite
1460 AutoInsuranceService/
1461 AutoInsurance.h
1462 AutoInsuranceImpl.componentType

1463 *Snippet 7-9: Contribution with ComponentType*

1464
1465 The SCDL for the AutoInsuranceService component of Snippet 7-9 is:

1466
1467 <component name="AutoInsuranceService">
1468 <implementation.cpp library="autoinsurance" path="bin/"
1469 class="AutoInsuranceImpl" />
1470 </component>

1471 *Snippet 7-10: Component Definition with Local ComponentType*

1472
1473 Since there is a one-to-one correspondence between implementations and componentTypes, when an
1474 implementation is shared between contributions, it is desirable to also share the componentType file.
1475 ComponentType files can be exported and imported in the same manner as executable files. The
1476 location of a `.componentType` file can be specified using the `@componentType` attribute of the
1477 `implementation.cpp` element.

1478
1479 <component name="UnderwritingService">
1480 <implementation.cpp library="rates" path="rates:bin/"
1481 class="UnderwritingImpl" componentType="rates:types/UnderwritingImpl"
1482 />
1483 </component>

1484 *Snippet 7-11: Component Definition with Imported ComponentType*

1485 **7.3 C++ Contribution Extensions**

1486 **7.3.1 Export.cpp**

1487 Snippet 7-12 shows the pseudo-schema for the C++ export element used to make an executable or
1488 componentType file visible outside of a contribution.

1489

```
1490     <!-- export.cpp schema snippet -->
1491     <export.cpp xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
1492         name="QName" path="string"? >
```

1493 *Snippet 7-12: Pseudo-schema for C++ Export Element*

1494

1495 The **export.cpp** element has the following **attributes**:

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

1500 **7.3.2 Import.cpp**

1501 Snippet 7-13 shows the pseudo-schema for the C++ import element used to reference an executable or
1502 componentType file that is outside of a contribution.

1503

```
1504     <!-- import.cpp schema snippet -->
1505     <import.cpp xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
1506         name="QName" location="string" >
```

1507 *Snippet 7-13: Pseudo-schema for C++ Import Element*

1508

1509 The **import.cpp** element has the following **attributes**:

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

8 C++ Interfaces

1516 A service interface can be defined by a C++ class which has only pure virtual public member functions.
1517 The class might additionally have private or protected member functions, but these are not part of the
1518 service interface.

1519 When mapping a C++ interface to WSDL or when comparing two C++ interfaces for compatibility, as
1520 defined by the Assembly specification [ASSEMBLY], it is necessary for an SCA implementation to
1521 determine the signature (return type, name, and the names and types of the parameters) of every
1522 member function of the class defining the service interface. An SCA implementation MUST translate a
1523 class to tokens as part of conversion to WSDL or compatibility testing. [CPP80001] Macros and typedefs
1524 in member function declarations might lead to portability problems. Complete member function
1525 declarations within a macro are discouraged. The processing of typedefs needs to be aware of the types
1526 that impact mapping to WSDL (see Table 9-1 and Table 9-2)

8.1 Types Supported in Service Interfaces

1527 Not all service interfaces support the complete set of the types available in C++.

8.1.1 Local Service

1530 Any fundamental or compound type defined by C++ can be used in the interface of a local service.

8.1.2 Remotable Service

1532 For a remotable service being called by another service the data exchange semantics is by-value. The
1533 return type and types of the parameters of a member function of a remotable service interface MUST be
1534 one of:

- 1535 • Any of the C++ types specified in Simple Content Binding. These types may be passed by-value, by-
1536 reference, or by-pointer. Unless the member function and client indicate that they allow by-reference
1537 semantics (see AllowsPassByReference), a copy will be explicitly created by the runtime for any
1538 parameters passed by-reference or by-pointer.
- 1539 • An SDO DataObjectPtr instance. This type may be passed by-value, by-reference, or by-pointer.
1540 Unless the member function and client indicate that they allow by-reference semantics (see
1541 AllowsPassByReference), a deep-copy of the DataObjectPtr will be created by the runtime for any
1542 parameters passed by-value, by-reference, or by-pointer. When by-reference semantics are allowed,
1543 the DataObjectPtr itself will be passed. [CPP80002]

8.2 Header Files

1545 A C++ header file used to define an interface MUST declare at least one class with:

- 1546 • At least one public member function.
- 1547 • All public member functions are pure virtual. [CPP80003]

9 WSDL to C++ and C++ to WSDL Mapping

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

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

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

Items in the JAX-WS WSDL to Java and Java to WSDL mapping that are not supported:

- JAX-WS style external binding. (See JAX-WS Sec. 2)
- MIME binding. (See JAX-WS Sec. 2.1.1)
- Holder classes. (See JAX-WS Sec. 2.3.3)
- Asynchronous mapping. (See JAX-WS Sec. 2.3.4)
- Generation of Service classes from WSDL. (See JAX-WS Sec. 2.7)
- Generation of WSDL from Service implementation classes (See JAX-WS Sec. 3.3)
- Templates when converting from C++ to WSDL (See JAX-WS Sec. 3.9)

General rules for the application of JAX-WS to C++.

- References to Java are considered references to C++.
- References to Java classes are considered references to C++ classes.
- References to Java methods are considered references to C++ member functions.
- References to Java interfaces are considered references to C++ classes which only define pure virtual member functions.

Major divergences from JAX-WS:

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

9.1 Augmentations for WSDL to C++ Mapping

An SCA implementation MUST map a WSDL portType to a remotable C++ interface definition.
[CPP100009]

9.1.1 Mapping WSDL targetNamespace to a C++ namespace

Since C++ does not define a standard convention for the use of namespaces, the SCA specification does not define an implicit mapping of WSDL targetNamespaces to C++ namespaces. A WSDL file 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. [CPP100001]

1587 9.1.2 Mapping WSDL Faults to C++ Exceptions

1588 WSDL operations that specify one or more <wsdl:fault> elements will produce a C++ member function
1589 that is annotated with an @WebThrows annotation listing a C++ exception class associated with each
1590 <wsdl:fault>.

1591 The C++ exception class associated with a fault will be generated based on the message that is
1592 associated with the <wsdl:fault> element, and in particular with the global element that the
1593 wsdl:fault/wsdl:message/@part indicates.

```
1595     <FaultException>(const char* message, const <FaultInfo>& faultInfo);  
1596     <FaultInfo> getFaultInfo() const;
```

1597 *Snippet 9-1: Fault Exception Class Member Functions*

1598
1599 Where <FaultException> is the name of the generated exception class, and where <FaultInfo> is
1600 the name of the C++ type representing the fault's global element type.

1601 9.1.2.1 Multiple Fault References

1602 If multiple operations within the same portType indicate that they throw faults that reference the same
1603 global element, an SCA implementation MUST generate a single C++ exception class with each C++
1604 member function referencing this class in its @WebThrows annotation. [CPP100002]

1605 9.1.3 Mapping of in, out, in/out parts to C++ member function parameters

1606 C++ diverges from the JAX-WS specification in it's handling of some parameter types, especially around
1607 how passing of out and in/out parameters are handled in the context of C++'s various pass-by styles.
1608 The following outlines an updated mapping for use with C++.

- 1609 • For unwrapped messages, an SCA implementation MUST map:
 - 1610 – **in** - the message part to a member function parameter, passed by const-reference.
 - 1611 – **out** - the message part to a member function parameter, passed by reference, or to the member
1612 function return type, returned by-value.
 - 1613 – **in/out** - the message part to a member function parameter, passed by reference. [CPP100003]
- 1614
- 1615 • For wrapped messages, an SCA implementation MUST map:
 - 1616 – **in** - the wrapper child to a member function parameter, passed by const-reference.
 - 1617 – **out** - the wrapper child to a member function parameter, passed by reference, or to the member
1618 function return type, returned by-value.
 - 1619 – **in/out** - the wrapper child to a member function parameter, passed by reference. [CPP100004]

1621 9.2 Augmentations for C++ to WSDL Mapping

1622 Where annotations are discussed as a means for an application to control the mapping to WSDL, an
1623 implementation-specific means of controlling the mapping can be used instead.

1624 An SCA implementation MUST map a C++ interface definition to WSDL as if it has a @WebService
1625 annotation with all default values on the class. [CPP100010]

1626 An application can customize the name of the portType and port using the @WebService annotation.

1627 **9.2.1 Mapping C++ namespaces to WSDL namespaces**

1628 Since C++ does not define a standard convention for the use of namespaces, the SCA specification does
1629 not define an implicit mapping of C++ namespaces to WSDL namespace URIs. The default
1630 targetNamespace is defined by the implementation. An SCA implementation SHOULD provide a
1631 mechanism for overriding the default targetNamespace. [CPP100005]

1632 **9.2.2 Parameter and return type classification**

1633 The classification of parameters and return types in C++ are determined based on how the value is
1634 passed into the function.

1635 An SCA implementation MUST map a method's return type as an **out** parameter, a parameter passed by-
1636 reference or by-pointer as an **in/out** parameter, and all other parameters, including those passed by-
1637 const-reference as **in** parameters. [CPP100006]

1638 An application can customize parameter classification using the @WebParam annotation.

1639 **9.2.3 C++ to WSDL Type Conversion**

1640 C++ types are mapped to WSDL and schema types based on the mapping described in Section Simple
1641 Content Binding.

1642 **9.2.4 Service-specific Exceptions**

1643 C++ classes that define a web service interface can indicate which faults they might throw using the
1644 @WebThrows annotation. @WebThrows lists the names of each C++ class that might be thrown as a
1645 fault from a particular member function. By default, no exceptions are mapped to operation faults.

1646 **9.3 SDO Data Binding**

1647 **9.3.1 Simple Content Binding**

1648 The translation of XSD simple content types to C++ types follows the convention defined in the SDO
1649 specification. Table 9-1 summarizes that mapping as it applies to SCA services.

1650

XSD Schema Type →	C++ Type	→ XSD Schema Type
anySimpleType	std::string	string
anyType	commonj::sdo::DataObject	anyType
anyURI	std::string	string
base64Binary	char*	string
boolean	bool	boolean
byte	int8_t	byte
date	std::string	string
dateTime	std::string	string
decimal	std::string	string
double	double	double
duration	std::string	string

ENTITIES	std::list<std::string>	IDREFS
ENTITY	std::string	string
float	float	float
gDay	std::string	string
gMonth	std::string	stirng
gMonthDay	std::string	string
gYear	std::string	string
gYearMonth	std::string	string
hexBinary	char*	string
ID	std::string	string
IDREF	std::string	string
IDREFS	std::list<std::string>	IDREFS
int	int32_t	int
integer	std::string	string
language	std::string	string
long	int64_t	long
Name	std::string	string
NCName	std::string	string
negativeInteger	std::string	string
NMOKEN	std::string	string
NMOKENS	std::list<std::string>	IDREFS
nonNegativeInteger	std::string	string
nonPositiveInteger	std::string	string
normalizedString	std::string	string
NOTATION	std::string	string
positiveInteger	std::string	string
QName	std::string	string
short	int16_t	short
string	std::string	string
time	std::string	string
token	std::string	string
unsignedByte	uint8_t	unsignedByte
unsignedInt	uint32_t	unsignedInt

unsignedLong	uint64_t	unsignedLong
unsignedShort	uint16_t	unsignedShort

1651 *Table 9-1: XSD simple type to C++ type mapping*

1652

1653 Table 9-2 defines the mapping of C++ types to XSD schema types that are not covered in Table 9-1.

1654

C++ Type →	XSD Schema Type
char	string
wchar_t	string
signed char	byte
unsigned char	unsignedByte
short	short
unsigned short	unsignedShort
int	int
unsigned int	unsignedInt
long	long
unsigned long	unsignedLong
long long	long
unsigned long long	unsignedLong
wchar_t *	string
long double	decimal
time_t	dateTime
struct tm	dateTime

1655 *Table 9-2: C++ type to XSD type mapping*

1656

1657 The C++ standard does not define value ranges for integer types so it is possible that on a platform
 1658 parameters or return values could have values that are out of range for the default XSD schema type. In
 1659 these circumstances, the mapping would need to be customized, using @WebParam or @WebResult if
 1660 supported, or some other implementation-specific mechanism.

1661 An SCA implementation MUST map simple types as defined in Table 9-1 and Table 9-2 by default.

1662 [CPP100008]

1663 9.3.2 Complex Content Binding

1664 Any XSD complex types are mapped to an instance of an SDO DataObject.

1665 10 Conformance

1666 The XML schema pointed to by the RDDL document at the SCA namespace URI, defined by the
1667 Assembly specification **[ASSEMBLY]** and extended by this specification, are considered to be
1668 authoritative and take precedence over the XML schema in this document.
1669 The XML schema pointed to by the RDDL document at the SCA C++ namespace URI, defined by this
1670 specification, is considered to be authoritative and takes precedence over the XML schema in this
1671 document.

1672 Normative code artifacts related to this specification are considered to be authoritative and take
1673 precedence over specification text.

1674 An SCA implementation MUST reject a composite file that does not conform to <http://docs.oasis-open.org/opencsa/sca/200912/sca-interface-cpp-1.1.xsd> or <http://docs.oasis-open.org/opencsa/sca/200912/sca-implementation-cpp-1.1.xsd>. **[CPP110001]**

1677 An SCA implementation MUST reject a componentType file that does not conform to <http://docs.oasis-open.org/opencsa/sca/200912/sca-interface-cpp-1.1.xsd>. **[CPP110002]**

1679 An SCA implementation MUST reject a contribution file that does not conform to <http://docs.oasis-open.org/opencsa/sca/200912/sca-contribution-cpp-1.1.xsd>. **[CPP110003]**

1681 An SCA implementation MUST reject a WSDL file that does not conform to <http://docs.oasis-open.org/opencsa/sca-c-cpp/cpp/200901/sca-wsdl-ext-cpp-1.1.xsd>. **[CPP110004]**

1683 10.1 Conformance Targets

1684 The conformance targets of this specification are:

- 1685 • **SCA implementations**, which provide a **runtime** for SCA components and potentially **tools** for
1686 authoring SCA artifacts, component descriptions and/or runtime operations.
- 1687 • **SCA documents**, which describe SCA artifacts, and specific **elements** within these documents.
- 1688 • **C++ component implementations**, which execute under the control of an SCA runtime.
- 1689 • **C++ files**, which define SCA service interfaces and implementations.
- 1690 • **WSDL files**, which define SCA service interfaces.

1691 10.2 SCA Implementations

1692 An implementation conforms to this specification if it meets these conditions:

- 1693 1. It MUST conform to the SCA Assembly Model Specification **[ASSEMBLY]** and the SCA Policy
1694 Framework **[POLICY]**.
- 1695 2. It MUST comply with all statements in Table F-1 and Table F-4 related to an SCA implementation,
1696 notably all mandatory statements have to be implemented.
- 1697 3. It MUST implement the SCA C++ API defined in section C++ API.
- 1698 4. It MUST implement the mapping between C++ and WSDL 1.1 **[WSDL11]** defined in WSDL to C++
1699 and C++ to WSDL Mapping.
- 1700 5. It MUST support <interface.cpp/> and <implementation.cpp/> elements as defined in Component
1701 Type and Component in composite and componentType documents.
- 1702 6. It MUST support <export.cpp/> and <import.cpp/> elements as defined in C++ Contributions in
1703 contribution documents.
- 1704 7. It MAY support source file annotations as defined in C++ SCA Annotations, C++ SCA Policy
1705 Annotations and C++ WSDL Mapping Annotations. If source file annotations are supported, the
1706 implementation MUST comply with all statements in Table F-2 related to an SCA implementation,
1707 notably all mandatory statements in that section have to be implemented.

- 1708 8. It MAY support WSDL extensions as defined in WSDL C++ Mapping Extensions. If WSDL
1709 extensions are supported, the implementation MUST comply with all statements in Table F-3 related
1710 to an SCA implementation, notably all mandatory statements in that section have to be implemented.

1711 **10.3 SCA Documents**

- 1712 An SCA document conforms to this specification if it meets these condition:
- 1713 1. It MUST conform to the SCA Assembly Model Specification [**ASSEMBLY**] and, if appropriate, the
1714 SCA Policy Framework [**POLICY**].
- 1715 2. If it is a composite document, it MUST conform to the <http://docs.oasis-open.org/opencsa/sca/200912/sca-interface-cpp-1.1.xsd> and <http://docs.oasis-open.org/opencsa/sca/200912/sca-implementation-cpp-1.1.xsd> schema and MUST comply with the
1716 additional constraints on the document contents as defined in Table F-1.
1717
1718 If it is a componentType document, it MUST conforms to the <http://docs.oasis-open.org/opencsa/sca/200912/sca-interface-cpp-1.1.xsd> schema and MUST comply with the
1719 additional constraints on the document contents as defined in Table F-1.
1720
1721 If it is a contribution document, it MUST conforms to the <http://docs.oasis-open.org/opencsa/sca/200912/sca-contribution-cpp-1.1.xsd> schema and MUST comply with the
1722 additional constraints on the document contents as defined in Table F-1.
1723
1724

1725 **10.4 C++ Files**

- 1726 A C++ files conforms to this specification if it meets the condition:
- 1727 1. It MUST comply with all statements in Table F-1, Table F-2 and Table F-4 related to C++ contents
1728 and annotations, notably all mandatory statements have to be satisfied.

1729 **10.5 WSDL Files**

- 1730 A WSDL file conforms to this specification if it meets these conditions:
- 1731 1. It is a valid WSDL 1.1 [**WSDL11**] document.
- 1732 2. It MUST comply with all statements in Table F-1, Table F-3 and Table F-4 related to WSDL contents
1733 and extensions, notably all mandatory statements have to be satisfied.

1734 A C++ SCA Annotations

1735 To allow developers to define SCA related information directly in source files, without having to separately
1736 author SCDL files, a set of annotations is defined. If annotations are supported by an implementation, the
1737 annotations defined here MUST be supported and MUST be mapped to SCDL as described. The SCA
1738 runtime MUST only process the SCDL files and not the annotations. [CPPA0001]

1739 The annotations are defined as C++ comments in interface and implementation header files, for example:

```
1740 // @Scope("stateless")
```

1742 *Snippet A-1: Example Annotation*

1743 A.1 Application of Annotations to C++ Program Elements

1744 In general an annotation immediately precedes the program element it applies to. If multiple annotations
1745 apply to a program element, all of the annotations SHOULD be in the same comment block. [CPPA0002]

- 1746 • Class

1747 The annotation immediately precedes the class.

1748 Example:

```
1749 // @Scope("composite")
1750 class LoanServiceImpl : public LoanService {
1751     ...
1752 };
```

1753 *Snippet A-2: Example Class Annotation*

- 1754 • Member function

1755 The annotation immediately precedes the member function.

1756 Example:

```
1757 class LoanService
1758 {
1759     public:
1760         // @OneWay
1761         virtual void reportEvent(int eventId) = 0;
1762         ...
1763 };
```

1764 *Snippet A-3: Example Member Function Annotation*

- 1765 • Data Member

1766 The annotation immediately precedes the data member.

1767 Example:

```
1768 // @Property(name="loanType", type="xsd:int")
1769 long loanType;
```

1770 *Snippet A-4: Example Data Member Annotation*

1771
1772 Annotations follow normal inheritance rules. An annotation on a base class or any element of a base
1773 class applies to any classes derived from the base class.

1774 A.2 Interface Header Annotations

1775 This section lists the annotations that can be used in the header file that defines a service interface.

1776 **A.2.1 @Interface**

1777 Annotation used to indicate a class defines an interface when multiple classes exist in a header file.

1778 **Corresponds to:** @class attribute of an *interface.cpp* element.

1779 **Format:**

```
1780 // @Interface
```

1781 *Snippet A-5: @Interface Annotation Format*

1782 **Applies to:** Class

1783 Example:

1784 Interface header:

```
1785 // @Interface
1786 class LoanService {
1787 ...
1788 };
```

1789

1790 Service definition:

```
1791 <service name="LoanService">
1792   <interface.cpp header="LoanService.h" class="LoanService" />
1793 </service>
```

1794 *Snippet A-6: Example of @Interface Annotation*

1795 **A.2.2 @Remotable**

1796 Annotation on service interface class to indicate that a service is remotable and implies an @Interface
1797 annotation applies as well. An SCA implementation MUST treat a class with an @WebService annotation
1798 specified as if a @Remotable annotation was specified. [CPPA0003]

1799 **Corresponds to:** @remutable="true" attribute of an *interface.cpp* element.

1800 **Format:**

```
1801 // @Remotable
```

1802 *Snippet A-7: @Remotable Annotation Format*

1803 The default is **false** (not remotable).

1804 **Applies to:** Class

1805 Example:

1806 Interface header:

```
1807 // @Remotable
1808 class LoanService {
1809 ...
1810 };
```

1811

1812 Service definition:

```
1813 <service name="LoanService">
1814   <interface.cpp header="LoanService.h" remotable="true" />
1815 </service>
```

1816 *Snippet A-8: Example of @Remotable Annotation*

1817 **A.2.3 @Callback**

1818 Annotation on a service interface class to specify the callback interface.

1819 **Corresponds to:** @callbackHeader and @callbackClass attributes of an *interface.cpp* element.

1820 **Format:**

```
1821 // @Callback(header="headerName", class="className")
```

1822 *Snippet A-9: @Callback Annotation Format*

1823 where

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

1826 **Applies to:** Class

1827 Example:

1828 Interface header:

```
1829 // @Callback(header="MyServiceCallback.h", class="MyServiceCallback")
1830 class MyService {
1831 public:
1832     virtual void someFunction( unsigned int arg ) = 0;
1833 };
```

1834

1835 Service definition:

```
1836 <service name="MyService">
1837   <interface.cpp header="MyService.h"
1838     callbackHeader="MyServiceCallback.h"
1839     callbackClass="MyServiceCallback" />
1840 </service>
```

1841 *Snippet A-10: Example of @Callback Annotation*

1842 A.2.4 @OneWay

1843 Annotation on an interface member function to indicate the member function is one way. The @OneWay
1844 annotation also affects the representation of a service in WSDL, see @OneWay.

1845 **Corresponds to:** @oneWay="true" attribute of *function* element of an *interface.cpp* element.

1846 **Format:**

```
1847 // @OneWay
```

1848 *Snippet A-11: @OneWay Annotation Format*

1849 The default is **false** (not OneWay).

1850 **Applies to:** Member function

1851 Example:

1852 Interface header:

```
1853 class LoanService
1854 {
1855 public:
1856 // @OneWay
1857     virtual void reportEvent(int eventId) = 0;
1858 ...
1859 };
```

1860

1861 Service definition:

```
1862 <service name="LoanService">
1863   <interface.cpp header="LoanService.h">
1864     <function name="reportEvent" oneWay="true" />
1865   </interface.cpp>
```

```
1866     </service>
1867 Snippet A-12: Example of @OneWay Annotation
```

1868 A.2.5 @Function

1869 Annotation on a interface member function to modify its representation in an SCA interface. An SCA
1870 implementation MUST treat a member function with a @WebFunction annotation specified as if
1871 @Function was specified with the operationName value of the @WebFunction annotation used as the
1872 name value of the @Function annotation and the exclude value of the @WebFunction annotation used as
1873 the exclude valued of the @Function annotation. [CPAA0004]

1874 **Corresponds to:** *function* or *callbackFunction* element of an *interface.cpp* element. If the class the
1875 function is a member of is being processed because it was identified via either a combination of
1876 *interface.cpp/@callbackHeader* and *interface.cpp/@callbackClass* or a @Callback annotation, then the
1877 @Function annotation corresponds to a *callbackFunction* element, otherwise it corresponds to a *function*
1878 element.

1879 **Format:**

```
1880 // @Function(name="operationName", exclude="true")
```

1881 Snippet A-13: @Function Annotation Format

1882 where

- **name : NCName (0..1)** – specifies the name of the operation. The default operation name is the function name.
- **exclude : boolean (0..1)** – specifies whether this member function is to be excluded from the SCA interface. Default is **false**.

1887 **Applies to:** Member function

1888 Example:

1889 Interface header:

```
1890 class LoanService
1891 {
1892     public:
1893     ...
1894     // @Function(exclude="true")
1895     virtual void reportEvent(int eventId) = 0;
1896     ...
1897 };
```

1898

1899 Service definition:

```
1900 <service name="LoanService">
1901   <interface.cpp header="LoanService.h">
1902     <function name="reportEvent" exclude="true" />
1903   </interface.cpp>
1904 </service>
```

1905 Snippet A-14: Example of @Function Annotation

1906 A.3 Implementation Header Annotations

1907 This section lists the annotations that can be used in the header file that defines a service
1908 implementation.

1909 A.3.1 @ComponentType

1910 Annotation used to indicate which class implements a componentType when multiple classes exist in an
1911 implementation file.

1912 **Corresponds to:** @class attribute of an *implementation.cpp* element.

1913 **Format:**

```
1914 // @ComponentType
```

1915 *Snippet A-15: @ComponentType Annotation Format*

1916 **Applies to:** Class

1917 Example:

1918 Implementation header:

```
1919 // @ComponentType
1920 class LoanServiceImpl : public LoanService {
1921 ...
1922 };
```

1923

1924 Component definition:

```
1925 <component name="LoanService">
1926   <implementation.cpp library="loan" class="LoanServiceImpl" />
1927 </component>
```

1928 *Snippet A-16: Example of @ComponentType Annotation*

A.3.2 @Scope

1930 Annotation on a service implementation class to indicate the scope of the service.

1931 **Corresponds to:** @scope attribute of an *implementation.cpp* element.

1932 **Format:**

```
1933 // @Scope("value")
```

1934 *Snippet A-17: @Scope Annotation Format*

1935 where

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

1938 **Applies to:** Class

1939 Example:

1940 Implementation header:

```
1941 // @Scope("composite")
1942 class LoanServiceImpl : public LoanService {
1943 ...
1944 };
```

1945

1946 Component definition:

```
1947 <component name="LoanService">
1948   <implementation.cpp library="loan" class="LoanServiceImpl"
1949     scope="composite" />
1950 </component>
```

1951 *Snippet A-18: Example of @Scope Annotation*

A.3.3 @EagerInit

1953 Annotation on a service implementation class to indicate the implantation is to be instantiated when its
1954 containing component is started.

1955 **Corresponds to:** @eagerInit="true" attribute of an *implementation.cpp* element.

1956 **Format:**

```
1957     // @EagerInit
```

1958 *Snippet A-19: @EagerInit Annotation Format*

1959 The default is **false** (the service is initialized lazily).

1960 **Applies to:** Class

1961 Example:

1962 Implementation header:

```
1963     // @EagerInit
1964     class LoanServiceImpl : public LoanService {
1965     ...
1966 }
```

1967

1968 Component definition:

```
1969 <component name="LoanService">
1970     <implementation.cpp library="loan" class="LoanServiceImpl"
1971         eagerInit="true" />
1972 </component>
```

1973 *Snippet A-20: Example of @EagerInit Annotation*

1974 **A.3.4 @AllowsPassByReference**

1975 Annotation on service implementation class or member function to indicate that a service or member
1976 function allows pass by reference semantics.

1977 **Corresponds to:** `@allowsPassByReference="true"` attribute of an *implementation.cpp* element or a
1978 *function* child element of an *implementation.cpp* element.

1979 **Format:**

```
1980     // @AllowsPassByReference
```

1981 *Snippet A-21: @AllowsPassByReference Annotation Format*

1982 The default is **false** (the service does not allow by reference parameters).

1983 **Applies to:** Class or Member function

1984 Example:

1985 Implementation header:

```
1986     // @AllowsPassByReference
1987     class LoanService {
1988     ...
1989 }
```

1990

1991 Component definition:

```
1992 <component name="LoanService">
1993     <implementation.cpp library="loan" class="LoanServiceImpl"
1994         allowsPassByReference="true" />
1995 </component>
```

1996 *Snippet A-22: Example of @AllowsPassByReference Annotation*

1997 **A.3.5 @Property**

1998 Annotation on a service implementation class data member to define a property of the service.

1999 **Corresponds to:** *property* element of a *componentType* element.

2000 **Format:**

```
2001     // @Property(name="propertyName", type="typeQName"
2002         //           default="defaultValue", required="true")
```

2003 *Snippet A-23: @Property Annotation Format*

2004 where

- ***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.
- ***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.
- ***required : boolean (0..1)*** - specifies whether a value has to be set in the component definition for this property. Default is *false*.
- ***default : <type> (0..1)*** - specifies a default value and is only needed if *required* is *false*,

2005 **Applies to:** DataMember

2006 Example:

2007 Implementation:

```
2008     // @Property(name="loanType", type="xsd:int")
2009     long loanType;
```

2010

2011

2012

2013

2014

2015

2016

2017

2018

2019 Component Type definition:

```
2020     <componentType ... >
2021         <service ... />
2022         <property name="loanType" type="xsd:int" />
2023     </componentType>
```

2024 *Snippet A-24: Example of @Property Annotation*

2025 **A.3.6 @Reference**

2026 Annotation on a service implementation class data member to define a reference of the service.

2027 **Corresponds to:** *reference* element of a *componentType* element.

2028 **Format:**

```
2029     // @Reference(name="referenceName", interfaceHeader="LoanService.h",
2030                 //           interfaceClass="LoanService", required="true")
```

2031 *Snippet A-25: @Reference Annotation Format*

2032 where

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

2033 If the annotated data member is a std::vector then the implied component type has a reference with a multiplicity of either 0..n or 1..n depending on the value of the @Reference *required* attribute – 1..n applies if *required=true*. Otherwise a multiplicity of 0..1 or 1..1 is implied.

2034

2035

2036

2037

2038

2039

2040

2041

2042 **Applies to:** Data Member

2043 Example:

```

2044     Implementation:
2045     // @Reference(interfaceHeader="LoanService.h" required="true")
2046     LoanServiceProxyPtr loanService;
2047
2048     // @Reference(interfaceHeader="LoanService.h" required="false")
2049     std::vector<LoanServiceProxyPtr> loanServices;
2050
2051     Component Type definition:
2052     <componentType ... >
2053         <service ... />
2054         <reference name="loanService" multiplicity="1..1">
2055             <interface.cpp header="LoanService.h" class="LoanService" />
2056         </reference>
2057         <reference name="loanServices" multiplicity="0..n">
2058             <interface.cpp header="LoanService.h" class="LoanService" />
2059         </reference>
2060     </componentType>

```

2061 *Snippet A-26: Example of @Reference Annotation*

2062 A.4 Base Annotation Grammar

```

2063 <annotation> ::= // @<baseAnnotation>
2064
2065 <baseAnnotation> ::= <name> [(<params>)]
2066
2067 <params> ::= <paramNameValue>[, <paramNameValue>]* | 
2068     <paramValue>[, <paramValue>]*
2069
2070 <paramNameValue> ::= <name>=<value>
2071
2072 <paramValue> ::= "<value>"
2073
2074 <name> ::= NCName
2075
2076 <value> ::= string

```

2077 *Snippet A-27: Base Annotation Grammar*

- 2078 • Adjacent string constants are concatenated
- 2079 • NCName is as defined by XML schema **[XSD]**
- 2080 • Whitespace including newlines between tokens is ignored.
- 2081 • Annotations with parameters can span multiple lines within a comment, and are considered complete
 2082 when the terminating ")" is reached.

2083 B C++ SCA Policy Annotations

2084 SCA provides facilities for the attachment of policy-related metadata to SCA assemblies, which influence
2085 how implementations, services and references behave at runtime. The policy facilities are described in
2086 [POLICY]. In particular, the facilities include Intents and Policy Sets, where intents express abstract,
2087 high-level policy requirements and policy sets express low-level detailed concrete policies.

2088 Policy metadata can be added to SCA assemblies through the means of declarative statements placed
2089 into Composite documents and into Component Type documents. These annotations are completely
2090 independent of implementation code, allowing policy to be applied during the assembly and deployment
2091 phases of application development.

2092 However, it can be useful and more natural to attach policy metadata directly to the code of
2093 implementations. This is particularly important where the policies concerned are relied on by the code
2094 itself. An example of this from the Security domain is where the implementation code expects to run
2095 under a specific security Role and where any service operations invoked on the implementation have to
2096 be authorized to ensure that the client has the correct rights to use the operations concerned. By
2097 annotating the code with appropriate policy metadata, the developer can rest assured that this metadata
2098 is not lost or forgotten during the assembly and deployment phases.

2099 The SCA C++ policy annotations provide the capability for the developer to attach policy information to
2100 C++ implementation code. The annotations provide both general facilities for attaching SCA Intents and
2101 Policy Sets to C++ code and annotations that deal with specific policy intents. Policy annotation can be
2102 used in header files for service interfaces or implementations.

2103 B.1 General Intent Annotations

2104 SCA provides the annotation **@Requires** for the attachment of any intent to a C++ class, to a C++
2105 interface or to elements within classes and interfaces such as member functions and data members.

2106 The **@Requires** annotation can attach one or multiple intents in a single statement. Each intent is
2107 expressed as a string. Intents are XML QNames, which consist of a Namespace URI followed by the
2108 name of the Intent. The precise form used is:

```
2109 " { " + Namespace URI + " } " + intentname
```

2110 *Snippet B-1: Intent Format*

2111
2112 Intents can be qualified, in which case the string consists of the base intent name, followed by a ".",
2113 followed by the name of the qualifier. There can also be multiple levels of qualification.

2114 This representation is quite verbose, so we expect that reusable constants will be defined for the
2115 namespace part of this string, as well as for each intent that is used by C++ code. SCA defines constants
2116 for intents such as the following:

```
2117  
2118 // @Define SCA_PREFIX "{http://docs.oasis-  
2119 open.org/ns/opencsa/sca/200912}"  
2120 // @Define CONFIDENTIALITY SCA PREFIX ## "confidentiality"  
2121 // @Define CONFIDENTIALITY_MESSAGE CONFIDENTIALITY ## ".message"
```

2122 *Snippet B-2: Example Intent Constants*

2123
2124 Notice that, by convention, qualified intents include the qualifier as part of the name of the constant,
2125 separated by an underscore. These intent constants are defined in the file that defines an annotation for
2126 the intent (annotations for intents, and the formal definition of these constants, are covered in a following
2127 section).

2128 Multiple intents (qualified or not) are expressed as separate strings within an array declaration.
2129 **Corresponds to:** @requires attribute of an *interface.cpp*, *implementation.cpp*, *function* or *callbackFunction*
2130 element.

2131 **Format:**

```
2132 // @Requires("qualifiedIntent" | {"qualifiedIntent" [, "qualifiedIntent"]})
```

2133 where

```
2134 qualifiedIntent ::= QName | QName.qualifier | QName.qualifier1.qualifier2
```

2135 *Snippet B-3: @Requires Annotation Format*

2136 **Applies to:** Class, Member Function

2137 Examples:

2138 Attaching the intents "confidentiality.message" and "integrity.message".

```
2139 // @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
```

2140 *Snippet B-4: Example @Requires Annotation*

2141

2142 A reference requiring support for confidentiality:

```
2143 class Foo {  
2144     ...  
2145     // @Requires(CONFIDENTIALITY)  
2146     // @Reference(interfaceHeader="SetBar.h")  
2147     void setBar(Bar* bar);  
2148     ...  
2149 }
```

2150 *Snippet B-5: @Requires Annotation applied with an @Reference Annotation*

2151

2152 Users can also choose to only use constants for the namespace part of the QName, so that they can add
2153 new intents without having to define new constants. In that case, this definition would instead look like
2154 this:

2155

```
2156 class Foo {  
2157     ...  
2158     // @Requires(SCA_PREFIX "confidentiality ")  
2159     // @Reference(interfaceHeader="SetBar.h")  
2160     void setBar(Bar* bar);  
2161     ...  
2162 }
```

2163 *Snippet B-6: @Requires Annotation Using Mixed Constants and Literals*

2164 **B.2 Specific Intent Annotations**

2165 In addition to the general intent annotation supplied by the @Requires annotation described above, there
2166 are C++ annotations that correspond to specific policy intents.

2167 The general form of these specific intent annotations is an annotation with a name derived from the name
2168 of the intent itself. If the intent is a qualified intent, qualifiers are supplied as an attribute to the annotation
2169 in the form of a string or an array of strings.

2170 For example, the SCA confidentiality intent described in General Intent Annotations using the
2171 @Requires(CONFIDENTIALITY) intent can also be specified with the specific @Confidentiality intent
2172 annotation. The specific intent annotation for the "integrity" security intent is:

2173

```
2174 // @Integrity
```

2175 Snippet B-7: Example Specific Intent Annotation

2176

2177 **Corresponds to:** @requires="<Intent>" attribute of an *interface.cpp*, *implementation.cpp*, *function* or
2178 *callbackFunction* element.

2179 **Format:**

2180 // @<Intent>[(qualifiers)]

2181 where Intent is an NCName that denotes a particular type of intent.

2182 Intent ::= NCName
2183 qualifiers ::= "qualifier" | {"qualifier" [, "qualifier"] }
2184 qualifier ::= NCName | NCName/qualifier

2185 Snippet B-8: @<Intent> Annotation Format

2186 **Applies to:** Class, Member Function – but see specific intents for restrictions

2187 Example:

2188 // @ClientAuthentication({"message", "transport"})

2189 Snippet B-9: Example @<Intent> Annotation

2190

2191 This annotation attaches the pair of qualified intents: *authentication.message* and *authentication.transport*
2192 (the sca: namespace is assumed in both of these cases – "http://docs.oasis-
2193 open.org/opencsa/ns/sca/200912").

2194

2195 The Policy Framework **[POLICY]** defines a number of intents and qualifiers. Security Interaction –
2196 Miscellaneous define the annotations for those intents.

2197 B.2.1 Security Interaction

Intent	Annotation
clientAuthentication	@ClientAuthentication
serverAuthentication	@ServerAuthentication
mutualAuthentication	@MutualAuthentication
confidentiality	@Confidentiality
integrity	@Integrity

2198 Table B-1: Security Interaction Intent Annotations

2199

2200 These three intents can be qualified with

- 2201 • transport
2202 • message

2203 B.2.2 Security Implementation

Intent	Annotation	Qualifiers
authorization	@Authorization	fine_grain

2204 Table B-2: Security Implementation Intent Annotations

2205 **B.2.3 Reliable Messaging**

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

2206 *Table B-3: Reliable Messaging Intent Annotations*

2207 **B.2.4 Transactions**

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

2208 *Table B-4: Transaction Intent Annotations*

2209 **B.2.5 Miscellaneous**

Intent	Annotation	Qualifiers
SOAP	@SOAP	v1_1 v1_2

2210 *Table B-5: Miscellaneous Intent Annotations*

2211 **B.3 Policy Set Annotations**

2212 The SCA Policy Framework uses Policy Sets to capture detailed low-level concrete policies (for example, a concrete policy is the specific encryption algorithm to use when encrypting messages when using a specific communication protocol to link a reference to a service).

2215 Policy Sets can be applied directly to C++ implementations using the **@PolicySets** annotation. The PolicySets annotation either takes the QName of a single policy set as a string or the name of two or more policy sets as an array of strings.

2218 **Corresponds to:** @policySets attribute of an *interface.cpp*, *implementation.cpp*, *function* or *callbackFuncion* element.

2220 **Format:**

```
2221 // @PolicySets("<policy set QName>" |
2222 //           { "<policy set QName>" [, "<policy set QName>"] })
```

2223 *Snippet B-10: @PolicySets Annotation Format*

2224 As for intents, PolicySet names are QNames – in the form of “{Namespace-URI}localPart”.

2225 **Applies to:** Class, Member Function,

2226 Example:

```
2227 // @Reference(name="helloService", interfaceHeader="helloService.h",
2228 //           required="true")
2229 // @PolicySets({ MY_NS "WS_Encryption_Policy",
2230 //                 MY_NS "WS_Authentication_Policy"})
2231 HelloService* helloService;
2232 ...
```

2233 *Snippet B-11: Example @PolicySets Annotation*

2234

2235 In this case, the Policy Sets WS_Encryption_Policy and WS_Authentication_Policy are applied, both
2236 using the namespace defined for the constant MY_NS.

2237 PolicySets satisfy intents expressed for the implementation when both are present, according to the rules
2238 defined in [POLICY].

2239 **B.4 Policy Annotation Grammar Additions**

```
2240 <annotation> ::= // @<baseAnnotation> | @<requiresAnnotation> | @<intentAnnotation> | @<policySetAnnotation>
2241                                     @<intents>
2242
2243 <requiresAnnotation> ::= Requires(<intents>)
2244
2245 <intents> ::= "<qualifiedIntent>" |
2246             {"<qualifiedIntent>", [<qualifiedIntent>]*})
2247
2248 <qualifiedIntent> ::= <intentName> | <intentName>.<qualifier> |
2249             <intentName>.<qualifier>.qualifier
2250
2251 <intentName> ::= {aAnyURI}NCName
2252
2253 <intentAnnotation> ::= <intent>[(<qualifiers>)]
2254
2255 <intent> ::= NCName [(param)]
2256
2257 <qualifiers> ::= "<qualifier>" | {"<qualifier>", "<qualifier>"}*
2258
2259 <qualifier> ::= NCName | NCName/<qualifier>
2260
2261 <policySetAnnotation> ::= policySets(<policysets>)
2262
2263 <policySets> ::= "<policySetName>" | {"<policySetName>", "<policySetName>"}*
2264
2265 <policySetName> ::= {aAnyURI}NCName
```

2266 *Snippet B-12: Annotation Grammar Additions for Policy Annotations*

- anyURI is as defined by XML schema [XSD]

2268 **B.5 Annotation Constants**

```
2269 <annotationConstant> ::= // @Define <identifier> <token string>
2270
2271 <identifier> ::= token
2272
2273 <token string> ::= "string" | "string"[ ## <token string>]
```

2274 *Snippet B-13: Annotation Constants Grammar*

- Constants are immediately expanded

2276 C C++ WSDL Mapping Annotations

2277 To allow developers to control the mapping of C++ to WSDL, a set of annotations is defined. If WSDL
2278 mapping annotations are supported by an implementation, the annotations defined here MUST be
2279 supported and MUST be mapped to WSDL as described. [CPPC0001]

2280 C.1 Interface Header Annotations

2281 C.1.1 @WebService

2282 Annotation on a C++ class indicating that it represents a web service. An SCA implementation MUST
2283 treat any instance of a @Remotable annotation and without an explicit @WebService annotation as if a
2284 @WebService annotation with no parameters was specified. [CPPC0002]

2285 **Corresponds to:** javax.jws.WebService annotation in the JAX-WS specification (7.11.1)

2286 **Format:**

```
2287 // @WebService(name="portTypeName", targetNamespace="namespaceURI",
2288 //                 serviceName="WSDLServiceName", portName="WSDLPortName")
```

2289 *Snippet C-1: @WebService Annotation Format*

2290 where:

- **name : NCName (0..1)** – specifies the name of the web service portType. The default is the name of the C++ class the annotation is applied to. The name of the associated binding is also determined by the portType. The binding name is the name of the portType suffixed with “Binding”.
- **targetNamespace : anyURI (0..1)** – specifies the target namespace for the web service. The default namespace is determined by the implementation.
- **serviceName : NCName (0..1)** – specifies the target name for the associated service. The default service name is the name of the C++ class suffixed with “Service”.
- **portName : NCName (0..1)** – specifies the name to be used for the associated WSDL port for the service. If portName is not specified, the name of the WSDL port is the name of the portType suffixed with “Port”. See [CPPF0042]

2301 **Applies to:** Class

2302 Example:

2303 Input C++ source file:

```
2304 // @WebService(name="StockQuote", targetNamespace="http://www.example.org/",
2305 //                 serviceName="StockQuoteService")
2306 class StockQuoteService {
2307 };
```

2308

2309 Generated WSDL file:

```
2310 <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
2311   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
2312   xmlns:tns="http://www.example.org/"
2313   xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
2314   targetNamespace="http://www.example.org/">
2315
2316   <portType name="StockQuote">
2317     <cpp:bindings>
2318       <cpp:class name="StockQuoteService"/>
2319     </cpp:bindings>
2320   </portType>
```

```

2321
2322     <binding name="StockQuoteServiceSoapBinding">
2323         <soap:binding style="document"
2324             transport="http://schemas.xmlsoap.org/soap/http"/>
2325     </binding>
2326
2327     <service name="StockQuoteService">
2328         <port name="StockQuotePort" binding="tns:StockQuoteServiceSoapBinding">
2329             <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
2330         </port>
2331     </service>
2332 </definitions>
```

2333 *Snippet C-2: Example @WebService Annotation*

2334 C.1.2 @WebFunction

2335 Annotation on a C++ member function indicating that it represents a web service operation. An SCA
 2336 implementation MUST treat a member function annotated with an @Function annotation and without an
 2337 explicit @WebFunction annotation as if a @WebFunction annotation with with an operationName value
 2338 equal to the name value of the @Function annotation, an exclude value equal to the exclude value of the
 2339 @Operation annotation and no other parameters was specified. [CPPC0009]

2340 **Corresponds to:** javax.jws.WebMethod annotation in the JAX-WS specification (7.11.2)

2341 **Format:**

```

2342 // @WebFunction(operationName="operation",    action="SOAPAction",
2343 //                 exclude="false")
```

2344 *Snippet C-3: @WebFunction Annotation Format*

2345 where:

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

2352 **Applies to:** Member function.

2353 Example:

2354 Input C++ source file:

```

2355 // @WebService(name="StockQuote", targetNamespace="http://www.example.org/"
2356 //               serviceName="StockQuoteService")
2357 class StockQuoteService {
2358
2359     // @WebFunction(operationName="GetLastTradePrice",
2360     //               action="urn:GetLastTradePrice")
2361     float getLastTradePrice(const std::string& tickerSymbol);
2362
2363     // @WebFunction(exclude=true)
2364     void setLastTradePrice(const std::string& tickerSymbol, float value);
2365 }
```

2366

2367 Generated WSDL file:

```

2368 <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
2369   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
2370   xmlns:tns="http://www.example.org/"
2371   xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
2372   targetNamespace="http://www.example.org/">
```

```

2373
2374     <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
2375         xmlns:tns="http://www.example.org/"
2376         attributeFormDefault="unqualified"
2377         elementFormDefault="unqualified"
2378         targetNamespace="http://www.example.org/">
2379     <xs:element name="GetLastTradePrice" type="tns:GetLastTradePrice"/>
2380     <xs:element name="GetLastTradePriceResponse"
2381         type="tns:GetLastTradePriceResponse"/>
2382     <xs:complexType name="GetLastTradePrice">
2383         <xs:sequence>
2384             <xs:element name="tickerSymbol" type="xs:string"/>
2385         </xs:sequence>
2386     </xs:complexType>
2387     <xs:complexType name="GetLastTradePriceResponse">
2388         <xs:sequence>
2389             <xs:element name="return" type="xs:float"/>
2390         </xs:sequence>
2391     </xs:complexType>
2392 </xs:schema>
2393
2394 <message name="GetLastTradePrice">
2395     <part name="parameters" element="tns:GetLastTradePrice">
2396     </part>
2397 </message>
2398
2399 <message name="GetLastTradePriceResponse">
2400     <part name="parameters" element="tns:GetLastTradePriceResponse">
2401     </part>
2402 </message>
2403
2404 <portType name="StockQuote">
2405     <cpp:bindings>
2406         <cpp:class name="StockQuoteService"/>
2407     </cpp:bindings>
2408     <operation name="GetLastTradePrice">
2409         <cpp:bindings>
2410             <cpp:memberFunction name="getLastTradePrice"/>
2411         </cpp:bindings>
2412         <input name="GetLastTradePrice" message="tns:GetLastTradePrice">
2413         </input>
2414         <output name="GetLastTradePriceResponse"
2415             message="tns:GetLastTradePriceResponse">
2416         </output>
2417     </operation>
2418 </portType>
2419
2420 <binding name="StockQuoteServiceSoapBinding">
2421     <soap:binding style="document"
2422         transport="http://schemas.xmlsoap.org/soap/http"/>
2423     <operation name="GetLastTradePrice">
2424         <soap:operation soapAction="urn:GetLastTradePrice" style="document"/>
2425         <input name="GetLastTradePrice">
2426             <soap:body use="literal"/>
2427         </input>
2428         <output name="GetLastTradePriceResponse">
2429             <soap:body use="literal"/>
2430         </output>
2431     </operation>
2432 </binding>
2433
2434 <service name="StockQuoteService">
2435     <port name="StockQuotePort" binding="tns:StockQuoteServiceSoapBinding">
2436         <soap:address location="REPLACE_WITH_ACTUAL_URL"/>

```

```
2437     </port>
2438   </service>
2439 </definitions>
```

2440 *Snippet C-4: Example @WebFunction Annotation*

2441 **C.1.3 @OneWay**

2442 Annotation on a C++ member function indicating that it represents a one-way request. The @OneWay
2443 annotation also affects the service interface, see @OneWay.

2444 **Corresponds to:** javax.jws.OneWay annotation in the JAX-WS specification (7.11.3)

2445 **Format:**

```
2446 // @OneWay
```

2447 *Snippet C-5: @OneWay Annotation Format*

2448 **Applies to:** Member function.

2449 Example:

2450 Input C++ source file:

```
2451 // @WebService(name="StockQuote", targetNamespace="http://www.example.org/")
2452 //           serviceName="StockQuoteService")
2453 class StockQuoteService {
2454
2455     // @WebFunction(operationName="SetTradePrice",
2456     //               action="urn:SetTradePrice")
2457     // @OneWay
2458     void setTradePrice(const std::string& tickerSymbol, float price);
2459 }
```

2460

2461 Generated WSDL file:

```
2462 <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
2463   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
2464   xmlns:tns="http://www.example.org/"
2465   xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
2466   targetNamespace="http://www.example.org/">
2467
2468   <xsschema xmlns:xs="http://www.w3.org/2001/XMLSchema"
2469     xmlns:tns="http://www.example.org/"
2470     attributeFormDefault="unqualified"
2471     elementFormDefault="unqualified"
2472     targetNamespace="http://www.example.org/">
2473     <xselement name="SetTradePrice" type="tns:SetTradePrice"/>
2474     <xsccomplexType name="SetTradePrice">
2475       <xssquence>
2476         <xselement name="tickerSymbol" type="xs:string"/>
2477         <xselement name="price" type="xs:float"/>
2478       </xssquence>
2479     </xsccomplexType>
2480   </xsschema>
2481
2482   <message name="SetTradePrice">
2483     <part name="parameters" element="tns:SetTradePrice">
2484     </part>
2485   </message>
2486
2487   <portType name="StockQuote">
2488     <cpp:bindings>
2489       <cpp:class name="StockQuoteService"/>
2490     </cpp:bindings>
2491     <operation name="SetTradePrice">
```

```

2492         <cpp:bindings>
2493             <cpp:memberFunction name="setTradePrice"/>
2494         </cpp:bindings>
2495         <input name="SetTradePrice" message="tns:SetTradePrice">
2496             </input>
2497         </operation>
2498     </portType>
2499
2500     <binding name="StockQuoteServiceSoapBinding">
2501         <soap:binding style="document"
2502             transport="http://schemas.xmlsoap.org/soap/http"/>
2503         <operation name="SetTradePrice">
2504             <soap:operation soapAction="urn:SetTradePrice" style="document"/>
2505             <input name="SetTradePrice">
2506                 <soap:body use="literal"/>
2507             </input>
2508         </operation>
2509     </binding>
2510
2511     <service name="StockQuoteService">
2512         <port name="StockQuotePort" binding="tns:StockQuoteServiceSoapBinding">
2513             <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
2514         </port>
2515     </service>
2516 </definitions>

```

2517 *Snippet C-6: Example @OneWay Annotation*

2518 C.1.4 @WebParam

2519 Annotation on a C++ member function indicating the mapping of a parameter to the associated input and
2520 output WSDL messages.

2521 **Corresponds to:** javax.jws.WebParam annotation in the JAX-WS specification (7.11.4)

2522 **Format:**

```

2523 // @WebParam(paramName=<"parameter", name="WSDLElement",
2524 //             targetNamespace="namespaceURI", mode="IN|OUT|INOUT",
2525 //             header="false", partName="WSDLPart", type="xsdType")

```

2526 *Snippet C-7: @WebParam Annotation Format*

2527 where:

- **paramName : NCName (1..1)** – specifies the name of the parameter that this annotation applies to. The value of the paramName of a @WebParam annotation MUST be the name of a parameter of the member function the annotation is applied to. [CPPC0004]
- **name : NCName (0..1)** – specifies the name of the associated WSDL part or element. The default value is the name of the parameter. If an @WebParam annotation is not present, and the parameter is unnamed, then a name of “argN”, where N is an incrementing value from 1 indicating the position of the parameter in the argument list, will be used.
- **targetNamespace : string (0..1)** – specifies the target namespace for the part. The default namespace is the namespace of the associated @WebService. The targetNamespace attribute is ignored unless the binding style is document, and the binding parameterStyle is bare. See @SOAPBinding.
- **mode : token (0..1)** – specifies whether the parameter is associated with the input message, output message, or both. The default value is determined by the passing mechanism for the parameter, see Parameter and return type classification.
- **header : boolean (0..1)** – specifies whether this parameter is associated with a SOAP header element. The default value is “false”.

- 2544 • ***partName : NCName (0..1)*** – specifies the name of the WSDL part associated with this item. The
 2545 default value is the value of name.
- 2546 • ***type : NCName (0..1)*** – specifies the XML Schema type of the WSDL part or element associated with
 2547 this parameter. The value of the type property of a @WebParam annotation MUST be one of the
 2548 simpleTypes defined in namespace <http://www.w3.org/2001/XMLSchema>. [CPPC0005] The default
 2549 type is determined by the mapping defined in Simple Content Binding.

2550 **Applies to:** Member function parameter.

2551 Example:

2552 Input C++ source file:

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

    // @WebFunction(operationName="GetLastTradePrice",
    //               action="urn:GetLastTradePrice")
    // @WebParam(paramName="tickerSymbol", name="symbol")
    float getLastTradePrice(const std::string& tickerSymbol);
};
```

2562

2563 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="symbol" 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>
  </portType>
```

```

2604 <operation name="GetLastTradePrice">
2605   <cpp:bindings>
2606     <cpp:memberFunction name="getLastTradePrice"/>
2607     <cpp:parameter name="tickerSymbol"
2608       part="tns:GetLastTradePrice/parameter"
2609       childElementName="symbol"/>
2610   </cpp:bindings>
2611   <input name="GetLastTradePrice" message="tns:GetLastTradePrice">
2612   </input>
2613   <output name="GetLastTradePriceResponse"
2614     message="tns:GetLastTradePriceResponse">
2615   </output>
2616 </operation>
2617 </portType>
2618
2619 <binding name="StockQuoteServiceSoapBinding">
2620   <soap:binding style="document"
2621     transport="http://schemas.xmlsoap.org/soap/http"/>
2622   <operation name="GetLastTradePrice">
2623     <soap:operation soapAction="urn:GetLastTradePrice" style="document"/>
2624     <input name="GetLastTradePrice">
2625       <soap:body use="literal"/>
2626     </input>
2627     <output name="GetLastTradePriceResponse">
2628       <soap:body use="literal"/>
2629     </output>
2630   </operation>
2631 </binding>
2632
2633 <service name="StockQuoteService">
2634   <port name="StockQuotePort" binding="tns:StockQuoteServiceSoapBinding">
2635     <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
2636   </port>
2637 </service>
2638 </definitions>
```

2639 *Snippet C-8: Example @WebParam Annotation*

2640 C.1.5 @WebResult

2641 Annotation on a C++ member function indicating the mapping of the member function's return type to the
2642 associated output WSDL message.

2643 **Corresponds to:** javax.jws.WebResult annotation in the JAX-WS specification (7.11.5)

2644 **Format:**

```

2645 // @WebResult(name=<"WSDLElement", targetNamespace="namespaceURI",
2646 //               header="false", partName="WSDLPart", type="xsdType")
```

2647 *Snippet C-9: @WebResult Annotation Format*

2648 where:

- **name : NCName (0..1)** – specifies the name of the associated WSDL part or element. The default value is “return”.
- **targetNamespace : string (0..1)** – specifies the target namespace for the part. The default namespace is the namespace of the associated @WebService. The targetNamespace attribute is ignored unless the binding style is document, and the binding parameterStyle is bare. See @SOAPBinding.
- **header : boolean (0..1)** – specifies whether the result is associated with a SOAP header element. The default value is “false”.
- **partName : NCName (0..1)** – specifies the name of the WSDL part associated with this item. The default value is the value of name.

- 2659 • ***type : NCName (0..1)*** – specifies the XML Schema type of the WSDL part or element associated with
 2660 this parameter. The value of the type property of a @WebResult annotation MUST be one of the
 2661 simpleTypes defined in namespace <http://www.w3.org/2001/XMLSchema>. [CPPC0006] The default
 2662 type is determined by the mapping defined in Simple Content Binding.

2663 **Applies to:** Member function return value.

2664 Example:

2665 Input C++ source file:

```
2666 // @WebService(name="StockQuote", targetNamespace="http://www.example.org/")
2667 //           serviceName="StockQuoteService")
2668 class StockQuoteService {
2669
2670     // @WebFunction(operationName="GetLastTradePrice",
2671     //               action="urn:GetLastTradePrice")
2672     // @WebResult(name="price")
2673     float getLastTradePrice(const std::string& tickerSymbol);
2674 }
```

2675

2676 Generated WSDL file:

```
2677 <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
2678   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
2679   xmlns:tns="http://www.example.org/"
2680   xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
2681   targetNamespace="http://www.example.org/">
2682
2683   <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
2684     xmlns:tns="http://www.example.org/"
2685     attributeFormDefault="unqualified"
2686     elementFormDefault="unqualified"
2687     targetNamespace="http://www.example.org/">
2688     <xs:element name="GetLastTradePrice" type="tns:GetLastTradePrice"/>
2689     <xs:element name="GetLastTradePriceResponse"
2690       type="tns:GetLastTradePriceResponse"/>
2691     <xs:complexType name="GetLastTradePrice">
2692       <xs:sequence>
2693         <xs:element name="tickerSymbol" type="xs:string"/>
2694       </xs:sequence>
2695     </xs:complexType>
2696     <xs:complexType name="GetLastTradePriceResponse">
2697       <xs:sequence>
2698         <xs:element name="price" type="xs:float"/>
2699       </xs:sequence>
2700     </xs:complexType>
2701   </xs:schema>
2702
2703   <message name="GetLastTradePrice">
2704     <part name="parameters" element="tns:GetLastTradePrice">
2705     </part>
2706   </message>
2707
2708   <message name="GetLastTradePriceResponse">
2709     <part name="parameters" element="tns:GetLastTradePriceResponse">
2710     </part>
2711   </message>
2712
2713   <portType name="StockQuote">
2714     <cpp:bindings>
2715       <cpp:class name="StockQuoteService"/>
2716     </cpp:bindings>
2717     <operation name="GetLastTradePrice">
2718       <cpp:bindings>
```

```

2719             <cpp:memberFunction name="getLastTradePrice"/>
2720         </cpp:bindings>
2721         <input name="GetLastTradePrice" message="tns:GetLastTradePrice">
2722             </input>
2723             <output name="GetLastTradePriceResponse"
2724                 message="tns:GetLastTradePriceResponse">
2725             </output>
2726         </operation>
2727     </portType>
2728
2729     <binding name="StockQuoteServiceSoapBinding">
2730         <soap:binding style="document"
2731             transport="http://schemas.xmlsoap.org/soap/http"/>
2732         <operation name="GetLastTradePrice">
2733             <soap:operation soapAction="urn:GetLastTradePrice" style="document"/>
2734             <input name="GetLastTradePrice">
2735                 <soap:body use="literal"/>
2736             </input>
2737             <output name="GetLastTradePriceResponse">
2738                 <soap:body use="literal"/>
2739             </output>
2740         </operation>
2741     </binding>
2742
2743     <service name="StockQuoteService">
2744         <port name="StockQuotePort" binding="tns:StockQuoteServiceSoapBinding">
2745             <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
2746         </port>
2747     </service>
2748 </definitions>

```

2749 *Snippet C-10: Example @WebResult Annotation*

2750 **C.1.6 @SOAPBinding**

2751 Annotation on a C++ member function indicating that it represents a web service operation.

2752 **Corresponds to:** javax.jws.SOAPBinding annotation in the JAX-WS specification (7.11.6)

2753 **Format:**

```

2754     // @SOAPBinding(style="DOCUMENT" | "RPC", use="LITERAL" | "ENCODED",
2755     //           parameterStyle="BARE" | "WRAPPED")

```

2756 *Snippet C-11: @SOAPBinding Annotation Format*

2757 where:

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

2762 **Applies to:** Class, Member function.

2763 Example:

2764 Input C++ source file:

```

2765     // @WebService(name="StockQuote", targetNamespace="http://www.example.org/"
2766     //           serviceName="StockQuoteService")
2767     // @SOAPBinding(style="RPC")
2768     class StockQuoteService {
2769     };

```

2770

2771 Generated WSDL file:

```

2772 <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
2773   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
2774   xmlns:tns="http://www.example.org/"
2775   xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
2776   targetNamespace="http://www.example.org/">
2777
2778   <portType name="StockQuote">
2779     <cpp:bindings>
2780       <cpp:class name="StockQuoteService"/>
2781     </cpp:bindings>
2782   </portType>
2783
2784   <binding name="StockQuoteServiceSoapBinding">
2785     <soap:binding style="document"
2786       transport="http://schemas.xmlsoap.org/soap/http"/>
2787   </binding>
2788
2789   <service name="StockQuoteService">
2790     <port name="StockQuotePort" binding="tns:StockQuoteServiceSoapBinding">
2791       <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
2792     </port>
2793   </service>
2794 </definitions>
```

2795 *Snippet C-12: Example @SOAPBinding Annotation*

2796 C.1.7 @WebFault

2797 Annotation on a C++ exception class indicating that it might be thrown as a fault by a web service
2798 function. A C++ class with a @WebFault annotation MUST provide a constructor that takes two
2799 parameters, a std::string and a type representing the fault information. Additionally, the class MUST
2800 provide a const member function "getFaultInfo" that takes no parameters, and returns the same type as
2801 defined in the constructor. [CPPC0007]

2802 **Corresponds to:** javax.xml.ws.WebFault annotation in the JAX-WS specification (7.2)

2803 **Format:**

```
2804 // @WebFault(name="WSDLElement", targetNamespace="namespaceURI")
```

2805 *Snippet C-13: @WebFault Annotation Format*

2806 where:

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

2810 **Applies to:** Class.

2811 Example:

2812 Input C++ source file:

```

2813 // @WebFault(name="UnknownSymbolFault",
2814 //             targetNamespace="http://www.example.org/")
2815 class UnknownSymbol {
2816   UnknownSymbol(const char* message,
2817                 const std::string& faultInfo);
2818
2819   std::string getFaultInfo() const;
2820 };
2821
2822 // @WebService(name="StockQuote", targetNamespace="http://www.example.org/")
2823 //             serviceName="StockQuoteService")
2824 class StockQuoteService {
```

```

2826     // @WebFunction(operationName="GetLastTradePrice",
2827     //           action="urn:GetLastTradePrice")
2828     // @WebThrows(faults="UnknownSymbol")
2829     float getLastTradePrice(const std::string& tickerSymbol);
2830 }

```

2831

2832 Generated WSDL file:

```

2833 <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
2834   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
2835   xmlns:tns="http://www.example.org/"
2836   xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
2837   targetNamespace="http://www.example.org/">
2838
2839   <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
2840     xmlns:tns="http://www.example.org/"
2841     attributeFormDefault="unqualified"
2842     elementFormDefault="unqualified"
2843     targetNamespace="http://www.example.org/">
2844     <xs:element name="GetLastTradePrice" type="tns:GetLastTradePrice"/>
2845     <xs:element name="GetLastTradePriceResponse"
2846       type="tns:GetLastTradePriceResponse"/>
2847     <xs:complexType name="GetLastTradePrice">
2848       <xs:sequence>
2849         <xs:element name="tickerSymbol" type="xs:string"/>
2850       </xs:sequence>
2851     </xs:complexType>
2852     <xs:complexType name="GetLastTradePriceResponse">
2853       <xs:sequence>
2854         <xs:element name="return" type="xs:float"/>
2855       </xs:sequence>
2856     </xs:complexType>
2857     <xs:element name="UnknownSymbolFault" type="xs:string"/>
2858   </xs:schema>
2859
2860   <message name="GetLastTradePrice">
2861     <part name="parameters" element="tns:GetLastTradePrice">
2862     </part>
2863   </message>
2864
2865   <message name="GetLastTradePriceResponse">
2866     <part name="parameters" element="tns:GetLastTradePriceResponse">
2867     </part>
2868   </message>
2869
2870   <message name="UnknownSymbol">
2871     <part name="parameters" element="tns:UnknownSymbolFault">
2872     </part>
2873   </message>
2874
2875   <portType name="StockQuote">
2876     <cpp:bindings>
2877       <cpp:class name="StockQuoteService"/>
2878     </cpp:bindings>
2879     <operation name="GetLastTradePrice">
2880       <cpp:bindings>
2881         <cpp:memberFunction name="getLastTradePrice"/>
2882       </cpp:bindings>
2883       <input name="GetLastTradePrice" message="tns:GetLastTradePrice">
2884       </input>
2885       <output name="GetLastTradePriceResponse"
2886             message="tns:GetLastTradePriceResponse">
2887       </output>
2888       <fault name="UnknownSymbol" message="tns:UnknownSymbol">

```

```

2889         </fault>
2890     </operation>
2891 </portType>
2892
2893     <binding name="StockQuoteServiceSoapBinding">
2894         <soap:binding style="document"
2895             transport="http://schemas.xmlsoap.org/soap/http"/>
2896         <operation name="GetLastTradePrice">
2897             <soap:operation soapAction="urn:GetLastTradePrice" style="document"/>
2898             <input name="GetLastTradePrice">
2899                 <soap:body use="literal"/>
2900             </input>
2901             <output name="GetLastTradePriceResponse">
2902                 <soap:body use="literal"/>
2903             </output>
2904             <fault>
2905                 <soap:fault name="UnknownSymbol" use="literal"/>
2906             </fault>
2907         </operation>
2908     </binding>
2909
2910     <service name="StockQuoteService">
2911         <port name="StockQuotePort" binding="tns:StockQuoteServiceSoapBinding">
2912             <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
2913         </port>
2914     </service>
2915 </definitions>
```

2916 *Snippet C-14: Example @WebFault Annotation*

2917 C.1.8 @WebThrows

2918 Annotation on a C++ class indicating which faults might be thrown by this class.

2919 **Corresponds to:** No equivalent in JAX-WS.

2920 **Format:**

```
2921 // @WebThrows(faults="faultMsg1[, "faultMsgn"]*)
```

2922 *Snippet C-15: @WebThrows Annotation Format*

2923 where:

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

2927 **Applies to:** Member function.

2928 Example:

2929 See @WebFault.

2930 D WSDL C++ Mapping Extensions

2931 A set of WSDL extensions are used to augment the conversion process from WSDL to C++. All of these
2932 extensions are defined in the namespace <http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901>.
2933 For brevity, all definitions of these extensions will be fully qualified, and all references to the “cpp” prefix
2934 are associated with the namespace above. If WSDL extensions are supported by an implementation, all
2935 the extensions defined here MUST be supported and MUST be mapped to C++ as described.
2936 [CPPD0001]

2937 D.1 <cpp:bindings>

2938 <cpp:bindings> is a container type which can be used as a WSDL extension. All other SCA wsdl
2939 extensions will be specified as children of a <cpp:bindings> element. A <cpp:bindings> element can be
2940 used as an extension to any WSDL type that accepts extensions.

2941 D.2 <cpp:class>

2942 <cpp:class> provides a mechanism for defining an alternate C++ class name for a WSDL construct.

2943 Format:

```
2944 <cpp:class name="xsd:string"/>
```

2945 *Snippet D-1: <cpp:class> Element Format*

2946 where:

- 2947 • **class/@name : NCName (1..1)** – specifies the name of the C++ class associated with this WSDL
2948 element.

2949 Applicable WSDL element(s):

- 2950 • wsdl:portType
- 2951 • wsdl:fault

2952 A <cpp:bindings/> element MUST NOT have more than one <cpp:class/> child element. [CPPD0002]

2953 Example:

2954 Input WSDL file:

```
2955 <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"  
2956        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"  
2957        xmlns:tns="http://www.example.org/"  
2958        xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"  
2959        targetNamespace="http://www.example.org/">  
2960  
2961        <portType name="StockQuote">  
2962           <cpp:bindings>  
2963              <cpp:class name="StockQuoteService"/>  
2964            </cpp:bindings>  
2965        </portType>  
2966 </definitions>
```

2967
2968 Generated C++ file:

```
2969 // @WebService(name="StockQuote", targetNamespace="http://www.example.org/"  
2970 //            serviceName="StockQuoteService")  
2971 class StockQuoteService {  
2972 };
```

2973 *Snippet D-2: Example <cpp:class> Element*

2974 D.3 <cpp:enableWrapperStyle>

2975 <cpp:enableWrapperStyle> indicates whether or not the wrapper style for messages is applied, when
2976 otherwise applicable. If false, the wrapper style will never be applied.

2977 Format:

```
2978     <cpp:enableWrapperStyle>value</cpp:enableWrapperStyle>
```

2979 *Snippet D-3: <cpp:enableWrapperStyle> Element Format*

2980 where:

- 2981 • **enableWrapperStyle/text() : boolean (1..1)** – specifies whether wrapper style is enabled or disabled
2982 for this element and any of its children. The default value is “true”.

2983 Applicable WSDL element(s):

- 2984 • wsdl:definitions
- 2985 • wsdl:portType – overrides a binding applied to wsdl:definitions
- 2986 • wsdl:portType/wsdl:operation – overrides a binding applied to wsdl:definitions or the enclosing
2987 wsdl:portType

2988 A <cpp:bindings/> element MUST NOT have more than one <cpp:enableWrapperStyle/> child element.
2989 [CPPD0003]

2990 Example:

2991 Input WSDL file:

```
2992 <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"  
2993     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"  
2994     xmlns:tns="http://www.example.org/"  
2995     xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"  
2996     targetNamespace="http://www.example.org/">  
2997  
2998     <xsschema xmlns:xss="http://www.w3.org/2001/XMLSchema"  
2999         xmlns:tns="http://www.example.org/"  
3000         attributeFormDefault="unqualified"  
3001         elementFormDefault="unqualified"  
3002         targetNamespace="http://www.example.org/">  
3003         <xselement name="GetLastTradePrice" type="tns:GetLastTradePrice"/>  
3004         <xselement name="GetLastTradePriceResponse"  
3005             type="tns:GetLastTradePriceResponse"/>  
3006         <xsccomplexType name="GetLastTradePrice">  
3007             <xsssequence>  
3008                 <xselement name="tickerSymbol" type="xs:string"/>  
3009             </xsssequence>  
3010         </xsccomplexType>  
3011         <xsccomplexType name="GetLastTradePriceResponse">  
3012             <xsssequence>  
3013                 <xselement name="return" type="xs:float"/>  
3014             </xsssequence>  
3015         </xsccomplexType>  
3016     </xsschema>  
3017  
3018     <message name="GetLastTradePrice">  
3019         <part name="parameters" element="tns:GetLastTradePrice">  
3020             </part>  
3021     </message>  
3022  
3023     <message name="GetLastTradePriceResponse">  
3024         <part name="parameters" element="tns:GetLastTradePriceResponse">  
3025             </part>  
3026     </message>  
3027  
3028     <portType name="StockQuote">
```

```

3029     <cpp:bindings>
3030         <cpp:class name="StockQuoteService"/>
3031             <cpp:enableWrapperStyle>false</cpp:enableWrapperStyle>
3032         <cpp:bindings>
3033             <operation name="GetLastTradePrice">
3034                 <cpp:bindings>
3035                     <cpp:memberFunction name="getLastTradePrice"/>
3036                 </cpp:bindings>
3037                 <input name="GetLastTradePrice" message="tns:GetLastTradePrice">
3038                 </input>
3039                 <output name="GetLastTradePriceResponse"
3040                     message="tns:GetLastTradePriceResponse">
3041                 </output>
3042             </operation>
3043         </portType>
3044     </definitions>

```

3045

3046 Generated C++ file:

```

3047 // @WebService(name="StockQuote", targetNamespace="http://www.example.org/"
3048 //               serviceName="StockQuoteService")
3049 class StockQuoteService {
3050
3051     // @WebFunction(operationName="GetLastTradePrice",
3052     //               action="urn:GetLastTradePrice")
3053     commonj::sdo::DataObjectPtr
3054         getLastTradePrice(commonj::sdo::DataObjectPtr parameters);
3055 }

```

3056 *Snippet D-4: Example <cpp:enableWrapperStyle> Element*

3057 D.4 <cpp:namespace>

3058 <cpp:namespace> specifies the name of the C++ namespace that the associated WSDL element (and
3059 any of its children) are created in.

3060 **Format:**

```

3061     <cpp:namespace name="namespaceURI"/>

```

3062 *Snippet D-5: <cpp:namespace> Element Format*

3063 where:

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

3066 **Applicable WSDL element(s):**

- wsdl:definitions

3068 A <cpp:bindings/> element MUST NOT have more than one <cpp:namespace/> child element.
3069 [CPPD0004]

3070 Example:

3071 Input WSDL file:

```

3072 <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
3073   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
3074   xmlns:tns="http://www.example.org/"
3075   xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
3076   targetNamespace="http://www.example.org/">
3077     <cpp:bindings>
3078       <cpp:namespace name="stock"/>
3079     </cpp:bindings>
3080
3081     <portType name="StockQuote">

```

```
3082     <cpp:bindings>
3083         <cpp:class name="StockQuoteService"/>
3084     </cpp:bindings>
3085     </portType>
3086 </definitions>
```

3087

3088 Generated C++ file:

```
3089 namespace stock
3090 {
3091     // @WebService(name="StockQuote", targetNamespace="http://www.example.org/")
3092     //     serviceName="StockQuoteService")
3093     // @WebService(name="StockQuote",
3094     class StockQuoteService {
3095     };
3096 }
```

3097 *Snippet D-6: Example <cpp:namespace> Element*

3098 D.5 <cpp:memberFunction>

3099 <cpp:memberFunction> specifies the name of the C++ member function that the associated WSDL
3100 operation is associated with.

3101 **Format:**

```
3102 <cpp:memberFunction name="myFunction"/>
```

3103 *Snippet D-7: <cpp:memberFunction> Element Format*

3104 where:

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

3107 **Applicable WSDL element(s):**

- wsdl:portType/wsdl:operation

3109 A <cpp:bindings/> element MUST NOT have more than one <cpp:memberFunction/> child element.
3110 [CPPD0005]

3111 Example:

3112 Input WSDL file:

```
3113 <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
3114   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
3115   xmlns:tns="http://www.example.org/"
3116   xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
3117   targetNamespace="http://www.example.org/">
3118
3119   <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
3120     xmlns:tns="http://www.example.org/"
3121     attributeFormDefault="unqualified"
3122     elementFormDefault="unqualified"
3123     targetNamespace="http://www.example.org/">
3124     <xsd:element name="GetLastTradePrice" type="tns:GetLastTradePrice"/>
3125     <xsd:element name="GetLastTradePriceResponse"
3126       type="tns:GetLastTradePriceResponse"/>
3127     <xsd:complexType name="GetLastTradePrice">
3128       <xsd:sequence>
3129         <xsd:element name="tickerSymbol" type="xsd:string"/>
3130       </xsd:sequence>
3131     </xsd:complexType>
3132     <xsd:complexType name="GetLastTradePriceResponse">
3133       <xsd:sequence>
3134         <xsd:element name="return" type="xsd:float"/>
```

```

3135         </xs:sequence>
3136     </xs:complexType>
3137   </xs:schema>
3138
3139   <message name="GetLastTradePrice">
3140     <part name="parameters" element="tns:GetLastTradePrice">
3141     </part>
3142   </message>
3143
3144   <message name="GetLastTradePriceResponse">
3145     <part name="parameters" element="tns:GetLastTradePriceResponse">
3146     </part>
3147   </message>
3148
3149   <portType name="StockQuote">
3150     <cpp:bindings>
3151       <cpp:class name="StockQuoteService"/>
3152     </cpp:bindings>
3153     <operation name="GetLastTradePrice">
3154       <cpp:bindings>
3155         <cpp:memberFunction name="getTradePrice"/>
3156       </cpp:bindings>
3157       <input name="GetLastTradePrice" message="tns:GetLastTradePrice">
3158       </input>
3159       <output name="GetLastTradePriceResponse"
3160             message="tns:GetLastTradePriceResponse">
3161       </output>
3162     </operation>
3163   </portType>
3164 </definitions>

```

3165

3166 Generated C++ file:

```

// @WebService(name="StockQuote", targetNamespace="http://www.example.org/"
//             serviceName="StockQuoteService")
class StockQuoteService {

    // @WebFunction(operationName="GetLastTradePrice",
    //               action="urn:GetLastTradePrice")
    float getTradePrice(const std::string& tickerSymbol);
}

```

3175 *Snippet D-8: Example <cpp:memberFunction> Element*

3176 D.6 <cpp:parameter>

3177 <cpp:parameter> specifies the name of the C++ member function parameter associated with a specific
3178 WSDL message part or wrapper child element.

3179 **Format:**

```

3180   <cpp:parameter name="CPPParameter" part="WSDLPart"
3181     childElementName="WSDLElement" type="CPPType"/>

```

3182 *Snippet D-9: <cpp:parameter> Element Format*

3183 where:

- 3184 • **parameter/@name : NCName (1..1)** – specifies the name of the C++ member function parameter
3185 associated with this WSDL operation. “return” is used to denote the return value.
- 3186 • **parameter/@part : string (1..1)** - an XPath expression identifying the wsdl:part of a wsdl:message.
- 3187 • **parameter/@childElementName : QName (1..1)** – specifies the qualified name of a child element of
3188 the global element identified by parameter/@part.

- 3189 • **parameter/@type : string (0..1)** – specifies the type of the parameter or struct member or return
 3190 type. The @type attribute of a <cpp:parameter/> element MUST be a C++ type specified in Simple
 3191 Content Binding. [CPPD0006] The default type is determined by the mapping defined in Simple
 3192 Content Binding.

3193 **Applicable WSDL element(s):**

- 3194 • wsdl:portType/wsdl:operation

3195 Example:

3196 Input WSDL file:

```

3197 <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"  

3198   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"  

3199   xmlns:tns="http://www.example.org/"  

3200   xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"  

3201   targetNamespace="http://www.example.org/">  

3202  

3203   <xsschema xmlns:xss="http://www.w3.org/2001/XMLSchema"  

3204     xmlns:tns="http://www.example.org/"  

3205     attributeFormDefault="unqualified"  

3206     elementFormDefault="unqualified"  

3207     targetNamespace="http://www.example.org/">  

3208     <xselement name="GetLastTradePrice" type="tns:GetLastTradePrice"/>  

3209     <xselement name="GetLastTradePriceResponse"  

3210       type="tns:GetLastTradePriceResponse"/>  

3211     <xsccomplexType name="GetLastTradePrice">  

3212       <xsssequence>  

3213         <xselement name="symbol" type="xs:string"/>  

3214       </xsssequence>  

3215     </xsccomplexType>  

3216     <xsccomplexType name="GetLastTradePriceResponse">  

3217       <xsssequence>  

3218         <xselement name="return" type="xs:float"/>  

3219       </xsssequence>  

3220     </xsccomplexType>  

3221   </xsschema>  

3222  

3223   <message name="GetLastTradePrice">  

3224     <part name="parameters" element="tns:GetLastTradePrice">  

3225     </part>  

3226   </message>  

3227  

3228   <message name="GetLastTradePriceResponse">  

3229     <part name="parameters" element="tns:GetLastTradePriceResponse">  

3230     </part>  

3231   </message>  

3232  

3233   <portType name="StockQuote">  

3234     <cpp:bindings>  

3235       <cpp:class name="StockQuoteService"/>  

3236     </cpp:bindings>  

3237     <operation name="GetLastTradePrice">  

3238       <cpp:bindings>  

3239         <cpp:memberFunction name="getLastTradePrice"/>  

3240         <cpp:parameter name="tickerSymbol"  

3241           part="tns:GetLastTradePrice/parameter"  

3242           childElementName="symbol"/>  

3243       </cpp:bindings>  

3244       <input name="GetLastTradePrice" message="tns:GetLastTradePrice">  

3245       </input>  

3246       <output name="GetLastTradePriceResponse"  

3247         message="tns:GetLastTradePriceResponse">  

3248       </output>  

3249     </operation>
  
```

```

3250     </portType>
3251
3252     <binding name="StockQuoteServiceSoapBinding">
3253         <soap:binding style="document"
3254             transport="http://schemas.xmlsoap.org/soap/http"/>
3255         <operation name="GetLastTradePrice">
3256             <soap:operation soapAction="urn:GetLastTradePrice" style="document"/>
3257             <input name="GetLastTradePrice">
3258                 <soap:body use="literal"/>
3259             </input>
3260             <output name="GetLastTradePriceResponse">
3261                 <soap:body use="literal"/>
3262             </output>
3263         </operation>
3264     </binding>
3265
3266     <service name="StockQuoteService">
3267         <port name="StockQuotePort" binding="tns:StockQuoteServiceSoapBinding">
3268             <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
3269         </port>
3270     </service>
3271 </definitions>

```

3272

3273 Generated C++ file:

```

3274 // @WebService(name="StockQuote", targetNamespace="http://www.example.org/"
3275 //               serviceName="StockQuoteService")
3276 class StockQuoteService {
3277
3278     // @WebFunction(operationName="GetLastTradePrice",
3279     //               action="urn:GetLastTradePrice")
3280     // @WebParam(paramName="tickerSymbol", name="symbol")
3281     float getLastTradePrice(const std::string& tickerSymbol);
3282 };

```

3283 *Snippet D-10: Example <cpp:parameter> Element*

3284 D.7 JAX-WS WSDL Extensions

3285 An SCA implementation MAY support the reading and interpretation of JAX-WS defined WSDL
 3286 extensions; however it MUST give precedence to the corresponding SCA WSDL extension if present.
 3287 Table D-1 is a list of JAX-WS WSDL extensions that MAY be interpreted and their corresponding SCA
 3288 WSDL extensions. [CPPD0007]

3289

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

3290 *Table D-1: Allowed JAX-WS Extensions*

3291 D.8 sca-wsdlext-cpp-1.1.xsd

```

3292 <?xml version="1.0" encoding="UTF-8"?>
```

```

3293 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3294   targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca-c-
3295   cpp/cpp/200901"
3296   xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
3297   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
3298   elementFormDefault="qualified">
3299
3300   <element name="bindings" type="cpp:BindingsType" />
3301   <complexType name="BindingsType">
3302     <choice minOccurs="0" maxOccurs="unbounded">
3303       <element ref="cpp:namespace" />
3304       <element ref="cpp:class" />
3305       <element ref="cpp:enableWrapperStyle" />
3306       <element ref="cpp:memberFunction" />
3307       <element ref="cpp:parameter" />
3308     </choice>
3309   </complexType>
3310
3311   <element name="namespace" type="cpp:NamespaceType" />
3312   <complexType name="NamespaceType">
3313     <attribute name="name" type="xsd:anyURI" use="required" />
3314   </complexType>
3315
3316   <element name="class" type="cpp:ClassType" />
3317   <complexType name="ClassType">
3318     <attribute name="name" type="xsd:NCName" use="required" />
3319   </complexType>
3320
3321   <element name="memberFunction" type="cpp:MemberFunctionType" />
3322   <complexType name="MemberFunctionType">
3323     <attribute name="name" type="xsd:NCName" use="required" />
3324   </complexType>
3325
3326   <element name="parameter" type="cpp:ParameterType" />
3327   <complexType name="ParameterType">
3328     <attribute name="part" type="xsd:string" use="required" />
3329     <attribute name="childElementName" type="xsd:QName"
3330       use="required" />
3331     <attribute name="name" type="xsd:NCName" use="required" />
3332     <attribute name="type" type="xsd:string" use="optional" />
3333   </complexType>
3334
3335   <element name="enableWrapperStyle" type="xsd:boolean" />
3336 </schema>
```

3337 *Snippet D-11: SCA C++ WSDL Extension Schema*

3338

E XML Schemas

3339

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

```

3340 <?xml version="1.0" encoding="UTF-8"?>
3341 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3342     targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912"
3343     xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200912"
3344     elementFormDefault="qualified">
3345
3346     <include schemaLocation="sca-core.xsd"/>
3347
3348     <element name="interface.cpp" type="sca:CPPInterface"
3349         substitutionGroup="sca:interface"/>
3350
3351     <complexType name="CPPInterface">
3352         <complexContent>
3353             <extension base="sca:Interface">
3354                 <sequence>
3355                     <element name="function" type="sca:CPPFunction"
3356                         minOccurs="0" maxOccurs="unbounded" />
3357                     <element name="callbackFunction" type="sca:CPPFunction"
3358                         minOccurs="0" maxOccurs="unbounded" />
3359                     <any namespace="#other" processContents="lax"
3360                         minOccurs="0" maxOccurs="unbounded"/>
3361                 </sequence>
3362                 <attribute name="header" type="string" use="required"/>
3363                 <attribute name="class" type="Name" use="required"/>
3364                 <attribute name="callbackHeader" type="string" use="optional"/>
3365                 <attribute name="callbackClass" type="Name" use="optional"/>
3366             </extension>
3367         </complexContent>
3368     </complexType>
3369
3370     <complexType name="CPPFunction">
3371         <sequence>
3372             <choice minOccurs="0" maxOccurs="unbounded">
3373                 <element ref="sca:requires"/>
3374                 <element ref="sca:policySetAttachment"/>
3375             </choice>
3376             <any namespace="#other" processContents="lax" minOccurs="0"
3377                 maxOccurs="unbounded" />
3378         </sequence>
3379         <attribute name="name" type="NCName" use="required"/>
3380         <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3381         <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3382         <attribute name="oneWay" type="boolean" use="optional"/>
3383         <attribute name="exclude" type="boolean" use="optional"/>
3384         <anyAttribute namespace="#other" processContents="lax"/>
3385     </complexType>
3386
3387 </schema>

```

3388

Snippet E-1: SCA <interface.cpp> Schema

3389

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

```

3390 <?xml version="1.0" encoding="UTF-8"?>
3391 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3392     targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912"

```

```

3393     xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200912"
3394     elementFormDefault="qualified">
3395
3396     <include schemaLocation="sca-core.xsd"/>
3397
3398     <element name="implementation.cpp" type="sca:CPPImplementation"
3399         substitutionGroup="sca:implementation" />
3400     <complexType name="CPPImplementation">
3401         <complexContent>
3402             <extension base="sca:Implementation">
3403                 <sequence>
3404                     <element name="function" type="sca:CPPImplementationFunction"
3405                         minOccurs="0" maxOccurs="unbounded" />
3406                     <any namespace="#other" processContents="lax"
3407                         minOccurs="0" maxOccurs="unbounded"/>
3408                 </sequence>
3409                 <attribute name="library" type="NCName" use="required"/>
3410                 <attribute name="header" type="NCName" use="required"/>
3411                 <attribute name="path" type="string" use="optional"/>
3412                 <attribute name="class" type="Name" use="optional"/>
3413                 <attribute name="componentType" type="string" use="optional"/>
3414                 <attribute name="scope" type="sca:CPPImplementationScope"
3415                     use="optional"/>
3416                 <attribute name="eagerInit" type="boolean" use="optional"/>
3417                 <attribute name="allowsPassByReference" type="boolean"
3418                     use="optional"/>
3419             </extension>
3420         </complexContent>
3421     </complexType>
3422
3423     <simpleType name="CPPImplementationScope">
3424         <restriction base="string">
3425             <enumeration value="stateless"/>
3426             <enumeration value="composite"/>
3427         </restriction>
3428     </simpleType>
3429
3430     <complexType name="CPPImplementationFunction">
3431         <sequence>
3432             <choice minOccurs="0" maxOccurs="unbounded">
3433                 <element ref="sca:requires"/>
3434                 <element ref="sca:policySetAttachment"/>
3435             </choice>
3436             <any namespace="#other" processContents="lax" minOccurs="0"
3437                 maxOccurs="unbounded" />
3438         </sequence>
3439         <attribute name="name" type="NCName" use="required"/>
3440         <attribute name="requires" type="sca:listOfQNames" use="optional"/>
3441         <attribute name="policySets" type="sca:listOfQNames" use="optional"/>
3442         <attribute name="allowsPassByReference" type="boolean"
3443             use="optional"/>
3444         <anyAttribute namespace="#other" processContents="lax"/>
3445     </complexType>
3446
3447 </schema>

```

3448 *Snippet E-2 : SCA <implementation.cpp> Schema*

3449 **E.3 sca-contribution-cpp-1.1.xsd**

```

3450     <?xml version="1.0" encoding="UTF-8"?>
3451     <schema xmlns="http://www.w3.org/2001/XMLSchema"
3452         targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912">

```

```
3453      xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200912"
3454      elementFormDefault="qualified">
3455
3456      <include schemaLocation="sca-contributions.xsd"/>
3457
3458      <element name="export.cpp" type="sca:CPPExport"
3459          substitutionGroup="sca:Export"/>
3460
3461      <complexType name="CPPExport">
3462          <complexContent>
3463              <attribute name="name" type="QName" use="required"/>
3464              <attribute name="path" type="string" use="optional"/>
3465          </complexContent>
3466      </complexType>
3467
3468      <element name="import.cpp" type="sca:CPPImport"
3469          substitutionGroup="sca:Import"/>
3470
3471      <complexType name="CPPImport">
3472          <complexContent>
3473              <attribute name="name" type="QName" use="required"/>
3474              <attribute name="location" type="string" use="required"/>
3475          </complexContent>
3476      </complexType>
3477
3478  </schema>
```

3479 *Snippet E-3: SCA <export.cpp> and <import.cpp> Schema*

3480

F Normative Statement Summary

3481

This section contains a list of normative statements for this specification.

Conformance ID	Description
[CPP20001]	A C++ implementation MUST implement all of the operation(s) of the service interface(s) of its componentType.
[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 @header attribute of an <interface.cpp> element contains more than one class, then the @class attribute MUST be specified for the <interface.cpp> element.
[CPP20006]	If the header file identified by the @callbackHeader attribute of an <interface.cpp> element contains more than one class, then the @callbackClass attribute MUST be specified for the <interface.cpp> element.
[CPP20007]	The @name attribute of a <function> child element of a <interface.cpp> MUST be unique amongst the <function> elements of that <interface.cpp>.
[CPP20008]	The @name attribute of a <callbackFunction> child element of a <interface.cpp> MUST be unique amongst the <callbackFunction> elements of that <interface.cpp>.
[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 @class attribute of the <implementation.cpp> element.
[CPP20010]	The @name attribute of a <function> child element of a <implementation.cpp> MUST be unique amongst the <function> elements of that <implementation.cpp>.
[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 member function.
[CPP20013]	An SCA runtime MAY run multiple threads in a single composite scoped implementation instance object.
[CPP20014]	The SCA runtime MAY use by-reference semantics when passing input parameters, return values or exceptions on calls to remotable services within the same system address space if both the service member function implementation and the client are marked "allows pass by reference".
[CPP20015]	The SCA runtime MUST use by-value semantics when passing input parameters, return values and exceptions on calls to remotable services within the same system address space if the service member function implementation is not marked "allows pass by reference" or the client is not marked "allows pass by reference".

Conformance ID	Description
[CPP20016]	If the header file identified by the <code>@header</code> attribute of an <code><interface.cpp></code> element contains function declarations that are not operations of the interface, then the functions that are not operations of the interface MUST be excluded using <code><function></code> child elements of the <code><interface.cpp></code> element with <code>@exclude="true"</code> .
[CPP20017]	If the header file identified by the <code>@callbackHeader</code> attribute of an <code><interface.cpp></code> element contains function declarations that are not operations of the callback interface, then the functions that are not operations of the callback interface MUST be excluded using <code><callbackFunction></code> child elements of the <code><interface.cpp></code> element with <code>@exclude="true"</code> .
[CPP20018]	An SCA runtime MUST NOT perform any synchronization of access to component implementations.
[CPP30001]	If a remotable interface is defined with a C++ class, an SCA implementation SHOULD map the interface definition to WSDL before generating the proxy for the interface.
[CPP30002]	For each reference of a component, an SCA implementation MUST generate a service proxy derived from <code>ServiceProxy</code> that contains the operations of the reference's interface definition.
[CPP30003]	An SCA runtime MUST include an asynchronous invocation member function for every operation of a reference interface with a <code>@requires="asyncInvocation"</code> intent applied either to the operation or the reference as a whole.
[CPP30004]	An SCA runtime MUST include a response class for every response message of a reference interface that can be returned by an operation of the interface with a <code>@requires="asyncInvocation"</code> intent applied either to the operation of the reference as a whole.
[CPP40001]	An operation marked as <code>oneWay</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.
[CPP40002]	For each service of a component that includes a bidirectional interface, an SCA implementation MUST generate a service proxy derived from <code>ServiceProxy</code> that contains the operations of the reference's callback interface definition.
[CPP40003]	If a service of a component that has a callback interface contains operations with a <code>@requires="asyncInvocation"</code> intent applied either to the operation of the reference as a whole, an SCA implementation MUST include asynchronous invocation member functions and response classes as described in Long Running Request-Response Operations.
[CPP70001]	The <code>@name</code> attribute of a <code><export.cpp></code> element MUST be unique amongst the <code><export.cpp></code> elements in a domain.
[CPP70002]	The <code>@name</code> attribute of a <code><import.cpp></code> child element of a <code><contribution></code> MUST be unique amongst the <code><import.cpp></code> elements in of that contribution.
[CPP80001]	An SCA implementation MUST translate a class to tokens as part of conversion to WSDL or compatibility testing.

Conformance ID	Description
[CPP80002]	<p>The return type and types of the parameters of a member function of a remotable service interface MUST be one of:</p> <ul style="list-style-type: none"> Any of the C++ types specified in Simple Content Binding. These types may be passed by-value, by-reference, or by-pointer. Unless the member function and client indicate that they allow by-reference semantics (see AllowsPassByReference), a copy will be explicitly created by the runtime for any parameters passed by-reference or by-pointer. An SDO <code>DataObjectPtr</code> instance. This type may be passed by-value, by-reference, or by-pointer. Unless the member function and client indicate that they allow by-reference semantics (see AllowsPassByReference), a deep-copy of the <code>DataObjectPtr</code> will be created by the runtime for any parameters passed by-value, by-reference, or by-pointer. When by-reference semantics are allowed, the <code>DataObjectPtr</code> itself will be passed.
[CPP80003]	<p>A C++ header file used to define an interface MUST declare at least one class with:</p> <ul style="list-style-type: none"> At least one public member function. All public member functions are pure virtual.
[CPP100001]	<p>A WSDL file might define a namespace using the <code><sca:namespace></code> 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.</p>
[CPP100002]	<p>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 <code>@WebThrows</code> annotation.</p>
[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]	<p>An SCA implementation SHOULD provide a mechanism for overriding the default targetNamespace.</p>
[CPP100006]	<p>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.</p>
[CPP100008]	<p>An SCA implementation MUST map simple types as defined in Table 9-1 and Table 9-2 by default.</p>

Conformance ID	Description
[CPP10009]	An SCA implementation MUST map a WSDL portType to a remotable C++ interface definition.
[CPP10010]	An SCA implementation MUST map a C++ interface definition to WSDL as if it has a @WebService annotation with all default values on the class.
[CPP11001]	An SCA implementation MUST reject a composite file that does not conform to http://docs.oasis-open.org/opencsa/sca/200912/sca-interface-cpp-1.1.xsd or http://docs.oasis-open.org/opencsa/sca/200912/sca-implementation-cpp-1.1.xsd .
[CPP11002]	An SCA implementation MUST reject a componentType file that does not conform to http://docs.oasis-open.org/opencsa/sca/200912/sca-interface-cpp-1.1.xsd .
[CPP11003]	An SCA implementation MUST reject a contribution file that does not conform to http://docs.oasis-open.org/opencsa/sca/200912/sca-contribution-cpp-1.1.xsd .
[CPP11004]	An SCA implementation MUST reject a WSDL file that does not conform to http://docs.oasis-open.org/opencsa/sca-c-cpp/cpp/200901/sca-wsdl-ext-cpp-1.1.xsd .

3482 Table F-1: SCA C++ Core Normative Statements

3483 **F.1 Annotation Normative Statement Summary**

3484 This section contains a list of normative statements related to source file annotations for this specification.

Conformance ID	Description
[CPPA0001]	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 as if a @Remotable annotation was specified.
[CPPA0004]	An SCA implementation MUST treat a member function with a @WebFunction annotation specified as if @Function was specified with the operationName value of the @WebFunction annotation used as the name value of the @Function annotation and the exclude value of the @WebFunction annotation used as the exclude value of the @Function annotation.
[CPPC0001]	If WSDL mapping 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 @Remotable annotation and without an explicit @WebService annotation as if a @WebService annotation with no parameters was specified.
[CPPC0004]	The value of the paramName of a @WebParam annotation MUST be the name of a parameter of the member function the annotation is applied to.
[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 .

Conformance ID	Description
[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.
[CPPC0009]	An SCA implementation MUST treat a member function annotated with an @Function annotation and without an explicit @WebFunction annotation as if a @WebFunction annotation with an operationName value equal to the name value of the @Function annotation, an exclude value equal to the exclude value of the @Operation annotation and no other parameters was specified.

3485 Table F-2: SCA C++ Annotation Normative Statements

3486 F.2 WSDL Extention Normative Statement Summary

3487 This section contains a list of normative statements related to WSDL extensions for this specification.

Conformance ID	Description
[CPPD0001]	If WSDL 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]	A <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 <cpp:parameter/> element MUST be a C++ type specified in Simple Content Binding.
[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. Table D-1 is a list of JAX-WS WSDL extensions that MAY be interpreted and their corresponding SCA WSDL extensions.

3488 Table F-3 SCA C++ WSDL Extension Normative Statements

3489 **F.3 JAX-WS Normative Statements**

3490 The JAX-WS 2.1 specification [JAXWS21] defines normative statements for various requirements defined
 3491 by that specification. Table F-4 outlines those normative statements which apply to the WSDL mapping
 3492 described in this specification.

Number	Conformance Point	Notes	Conformance ID
2.1	WSDL 1.1 support	[A]	[CPPF0001]
2.2	Customization required	<p>[CPPD0001]</p> <p>The reference to the JAX-WS binding language is treated as a reference to the C++ WSDL extensions defined in section WSDL C++ Mapping Extensions.</p>	
2.3	Annotations on generated classes		[CPPF0002]
2.4	Definitions mapping	[CPP100001]	
2.5	WSDL and XML Schema import directives		[CPPF0003]
2.6	Optional WSDL extensions		[CPPF0004]
2.7	SEI naming		[CPPF0005]
2.8	javax.jws.WebService required	<p>[B]</p> <p>References to javax.jws.WebService in the conformance statement are treated as the C++ annotation @WebService.</p>	[CPPF0006]
2.10	Method naming		[CPPF0007]
2.11	javax.jws.WebMethod required	<p>[A], [B]</p> <p>References to javax.jws.WebMethod in the conformance statement are treated as the C++ annotation @WebFunction.</p>	[CPPF0008]
2.12	Transmission primitive support		[CPPF0009]
2.13	Using javax.jws.OneWay	<p>[A], [B]</p> <p>References to javax.jws.OneWay in the conformance statement are treated as the C++ annotation @OneWay.</p>	[CPPF0010]
2.14	Using javax.jws.SOAPBinding	<p>[A], [B]</p> <p>References to javax.jws.SOAPBinding in the conformance statement are treated as the C++ annotation @SOAPBinding.</p>	[CPPF0011]
2.15	Using javax.jws.WebParam	<p>[A], [B]</p> <p>References to javax.jws.WebParam in the conformance statement are treated as the C++ annotation @WebParam.</p>	[CPPF0012]

Number	Conformance Point	Notes	Conformance ID
2.16	Using javax.jws.WebResult	[A], [B] References to javax.jws.WebResult in the conformance statement are treated as the C++ annotation @WebResult.	[CPPF0013]
2.18	Non-wrapped parameter naming		[CPPF0014]
2.19	Default mapping mode		[CPPF0015]
2.20	Disabling wrapper style	[B] References to jaxws:enableWrapperStyle in the conformance statement are treated as the WSDL extension cpp:enableWrapperStyle.	[CPPF0016]
2.21	Wrapped parameter naming		[CPPF0017]
2.22	Parameter name clash	[A]	[CPPF0018]
2.38	javax.xml.ws.WebFault required	[B] References to javax.jws.WebFault in the conformance statement are treated as the C++ annotation @WebFault.	[CPPF0019]
2.39	Exception naming		[CPPF0020]
2.40	Fault equivalence	[CPP100002]	
2.42	Required WSDL extensions	MIME Binding not necessary	[CPPF0022]
2.43	Unbound message parts	[A]	[CPPF0023]
2.44	Duplicate headers in binding		[CPPF0024]
2.45	Duplicate headers in message		[CPPF0025]
3.1	WSDL 1.1 support	[A]	[CPPF0026]
3.2	Standard annotations	[A] [CPPC0001]	
3.3	Java identifier mapping	[A]	[CPPF0027]
3.4	Method name disambiguation	[A] References to javax.jws.WebMethod in the conformance statement are treated as the C++ annotation @WebFunction.	[CPPF0028]
3.6	WSDL and XML Schema import directives		[CPPF0029]
3.8	portType naming		[CPPF0030]

Number	Conformance Point	Notes	Conformance ID
3.9	Inheritance flattening	[A]	[CPPF0044]
3.10	Inherited interface mapping		[CPPF0045]
3.11	Operation naming		[CPPF0031]
3.12	One-way mapping	[B] References to javax.jws.OneWay in the conformance statement are treated as the C++ annotation @OneWay.	[CPPF0032]
3.13	One-way mapping errors		[CPPF0033]
3.15	Parameter classification	[CPP100006]	
3.16	Parameter naming		[CPPF0035]
3.17	Result naming		[CPPF0036]
3.18	Header mapping of parameters and results	References to javax.jws.WebParam in the conformance statement are treated as the C++ annotation @WebParam. References to javax.jws.WebResult in the conformance statement are treated as the C++ annotation @WebResult.	[CPPF0037]
3.27	Binding selection	References to the BindingType annotation are treated as references to SOAP related intents defined by [POLICY].	[CPPF0039]
3.28	SOAP binding support	[A]	[CPPF0040]
3.29	SOAP binding style required		[CPPF0041]
3.31	Port selection		[CPPF0042]
3.32	Port binding	References to the BindingType annotation are treated as references to SOAP related intents defined by [POLICY].	[CPPF0043]

3493 [A] All references to Java in the conformance point are treated as references to C++.

3494 [B] Annotation generation is only necessary if annotations are supported by an SCA implementation.

3495 *Table F-4: JAX-WS Normative Statements that are Applicable to SCA C++*

3496 F.3.1 Ignored Normative Statements

Number	Conformance Point
2.9	javax.xml.bind.XmlSeeAlso required
2.17	use of JAXB annotations
2.23	Using javax.xml.ws.RequestWrapper
2.24	Using javax.xml.ws.ResponseWrapper

Number	Conformance Point
2.25	Use of Holder
2.26	Asynchronous mapping required
2.27	Asynchronous mapping option
2.28	Asynchronous method naming
2.29	Asynchronous parameter naming
2.30	Failed method invocation
2.31	Response bean naming
2.32	Asynchronous fault reporting
2.33	Asynchronous fault cause
2.34	JAXB class mapping
2.35	JAXB customization use
2.36	JAXB customization clash
2.37	javax.xml.ws.wsaddressing.W3CEndpointReference
2.41	Fault Equivalence
2.46	Use of MIME type information
2.47	MIME type mismatch
2.48	MIME part identification
2.49	Service superclass required
2.50	Service class naming
2.51	javax.xml.ws.WebServiceClient required
2.52	Default constructor required
2.53	2 argument constructor required
2.54	Failed getPort Method
2.55	javax.xml.ws.WebEndpoint required
3.5	Package name mapping
3.7	Class mapping
3.14	use of JAXB annotations
3.19	Default wrapper bean names
3.20	Default wrapper bean package
3.21	Null Values in rpc/literal
3.24	Exception naming
3.25	java.lang.RuntimeExceptions and java.rmi.RemoteExceptions

Number	Conformance Point
3.26	Fault bean name clash
3.30	Service creation

3497 *Table F-5: JAX-WS Normative Statements that Are Not Applicable to SCA C++*

3498

G Migration

3499

To aid migration of an implementation or clients using an implementation based the version of the Service Component Architecture for C++ defined in [OSOA SCA C++ Client and Implementation V1.00](#), this

3500 identifies the relevant changes to APIs, annotations, or behavior defined in V1.00.

3501

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

3502

The `<method/>` child element of `<interface.cpp/>` and the `<method/>` child element of

3503 `<implementation.cpp/>` have both been renamed to `<function/>` to be consistent with C++ terminology.

3505

H Acknowledgements

3506 The following individuals have participated in the creation of this specification and are gratefully
3507 acknowledged:

3508 **Participants:**

3509

Participant Name	Affiliation
Bryan Aupperle	IBM
Andrew Borley	IBM
Jean-Sebastien Delfino	IBM
Mike Edwards	IBM
David Haney	Individual
Mark Little	Red Hat
Jeff Mischkinsky	Oracle Corporation
Peter Robbins	IBM

3510

I Revision History

3511

[optional; should not be included in OASIS Standards]

3512

Revision	Date	Editor	Changes Made
			•

3513