



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

**Committee Draft 01**

**20 March 2008**

**Specification URIs:**

**This Version:**

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

**Previous Version:**

N/A

**Latest Version:**

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

**Technical Committee:**

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

**Chair:**

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

**Editors:**

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

**Related work:**

This specification replaces or supercedes:

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

This specification is related to:

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

**Declared XML Namespace(s):**

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

**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 component implemented in C 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® 2005, 2008. All Rights Reserved.

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

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

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

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

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

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

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

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

---

# Table of Contents

1	Introduction .....	7
1.1	Terminology .....	7
1.2	Normative References .....	7
1.3	Non-Normative References .....	7
2	Basic Component Implementation Model .....	8
2.1	Implementing a Service .....	8
2.1.1	Implementing a Remotable Service .....	9
2.1.2	Implementing a Local Service .....	9
2.2	Conversational and Non-Conversational services .....	10
2.3	Component and Implementation Scopes .....	10
2.3.1	Stateless scope .....	11
2.3.2	Request scope .....	11
2.3.3	Composite scope .....	11
2.3.4	Conversation scope .....	11
2.4	Implementing a Configuration Property .....	12
2.5	Component Type and Component .....	12
2.5.1	Interface.c .....	13
2.5.2	Function and CallbackFunction .....	14
2.5.3	Implementation.c .....	15
2.5.4	Implementation Function .....	16
2.6	Implementing a Service with a Program .....	16
3	Basic Client Model .....	18
3.1	Accessing Services from Component Implementations .....	18
3.2	Accessing Services from non-SCA component implementations .....	19
3.3	Calling Service Operations .....	19
4	Conversational Services .....	20
4.1	Conversational Client .....	20
4.2	Conversational Service Provider .....	20
4.3	Operations that End the Conversation .....	21
4.4	Conversation Lifetime Summary .....	22
4.5	Application Specified Conversation IDs .....	22
4.6	Accessing Conversation IDs from Clients .....	22
5	Asynchronous Programming .....	24
5.1	Non-blocking Calls .....	24
5.2	Callbacks .....	24
5.2.1	Stateful Callbacks .....	25
5.2.2	Stateless Callbacks .....	26
5.2.3	Implementing Multiple Bidirectional Interfaces .....	26
5.2.4	Customizing the Callback Identity .....	26
6	Error Handling .....	27
7	C API .....	28
7.1	Synchronous Programming .....	28
7.2	Program-Based Implementation Support .....	30

7.3	Asynchronous Programming	30
7.4	Conversational Services	31
8	C to WSDL Mapping	32
8.1	Parameter and Return Type mappings	32
8.1.1	Built-in type mappings	32
8.1.2	Binary data mapping	33
8.1.3	Array mapping	33
8.1.4	Multi-dimensional array mapping	34
8.1.5	Pointer mapping	34
8.1.6	Struct mapping	35
8.1.7	Enum mapping	36
8.1.8	Union mapping	37
8.1.9	Typedef mapping	37
8.1.10	Pre-processor mapping	37
8.1.11	Nesting types	37
8.1.12	SDO mapping	38
8.1.13	void * mapping	39
8.1.14	Included types	39
8.2	Header mapping	39
8.3	Function mapping	41
8.3.1	Unnamed parameters	41
8.3.2	The return type	41
8.3.3	The void return type	42
8.3.4	No Parameters Specified	43
9	WSDL to C Mapping	44
9.1	Type Mapping	44
9.1.1	Simple type mapping	44
9.1.2	Complex type and group mapping	48
9.2	Message Mapping	50
9.3	Part Mapping	50
9.4	Operation Mapping	50
9.4.1	Operation Elements <input>, <output> and <fault>	50
9.4.2	One-way Operation	54
9.4.3	Request-Response Operation	55
9.5	PortType Mapping	55
9.6	Name Mapping	56
10	Packaging	57
10.1	Composite Packaging	57
11	Types Supported in Service Interfaces	58
11.1	Local service	58
11.2	Remotable service	58
12	Restrictions on C header files	59
13	Conformance	60
A	C Annotations	61
A.1	Application of Annotations to C Program Elements	61

A.2 Interface Header Annotations	62
A.2.1 @Interface	62
A.2.2 @Operation	62
A.2.3 @Remotable	63
A.2.4 @Callback	64
A.2.5 @OneWay	64
A.2.6 @Conversational	65
A.2.7 @EndsConversation	65
A.3 Implementation Annotations	66
A.3.1 @ComponentType	66
A.3.2 @Service	66
A.3.3 @Reference	67
A.3.4 @Property	68
A.3.5 @Scope	68
A.3.6 @Init	69
A.3.7 @Destroy	69
A.3.8 @EagerInit	70
A.3.9 @AllowsPassByReference	70
A.3.10 @ConversationAttributes	71
B Policy Annotations for C	72
B.1 General Intent Annotations	72
B.2 Specific Intent Annotations	73
B.3 Application of Intent Annotations	74
B.4 Policy Annotation Scope	74
B.5 Relationship of Declarative And Annotated Intents	76
B.6 Policy Set Annotations	76
B.7 Security Policy Annotations	76
B.7.1 Security Interaction Policy	76
B.7.2 Security Implementation Policy	77
C XML Schemas	81
C.1 sca-interface-c-1.1-schema.xsd	81
C.2 sca-implementation-c-1.1-schema.xsd	81
D Migration	83
D.1 Annotations related to conversations	83
E Acknowledgements	84
F Non-Normative Text	85
G Revision History	86

---

# 1 Introduction

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

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

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

## 1.1 Terminology

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

## 1.2 Normative References

- [RFC2119] S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, <http://www.ietf.org/rfc/rfc2119.txt>, IETF RFC 2119, March 1997.
- [ASSEMBLY] M. Beisiegel, et al., *Service Component Architecture Assembly Model Specification Version 1.1*, <http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec.pdf>, OASIS Service Component Architecture Assembly Model Specification Version 1.1, XXX 2008
- [POLICY] J. Anderson, et al., *SCA Policy Framework Version 1.1*, <http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec.pdf>, OASIS SCA Policy Framework Version 1.1, XXX 2008
- [SDO21] B. Aupperle, et al., *Service Data Objects For C Specification*, [http://www.osoa.org/download/attachments/36/SDO\\_Specification\\_C\\_V2.1.pdf](http://www.osoa.org/download/attachments/36/SDO_Specification_C_V2.1.pdf), SDO 2.1, September 2007.
- [WSDL11] E. Christensen, et al., *Web Service Description Language (WSDL)*, <http://www.w3.org/TR/wsdl>, W3C Note Web Service Description Language (WSDL), March 2001
- [WSDL20] R. Chinnici, et al., *Web Service Description Language (WSDL) Version 2.0 Part 1: Core Language*, <http://www.w3.org/TR/wsdl20/>, W3C Web Service Description Language (WSDL) Version 2.0 Part 1: Core Language, June 2007

## 1.3 Non-Normative References

---

## 2 Basic Component Implementation Model

36

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

40

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

### 2.1 Implementing a Service

44 A component implementation based on a set of C functions (a **C implementation**) provides one or more  
45 services.

46

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

- 48 • the declaration of the C functions implementing the services
- 49 • a WSDL 1.1 portType **[WSDL11]**
- 50 • a WSDL 2.0 interface **[WSDL20]**

51 If function declarations are used to define the interface, they will typically be placed in a separate header  
52 file. This is the service interface. The C component implementation **MUST** implement the operations of  
53 the service interface.

54

55 The following snippets show the C service interface and the C functions of a C implementation.

56

57 Service interface.

58

```
59 /* LoanService interface */  
60 char approveLoan(long customerNumber, long loanAmount);
```

61

62 Implementation.

```
63 #include "LoanService.h"  
64  
65 char approveLoan(long customerNumber, long loanAmount)  
66 {  
67     ...  
68 }
```

69

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

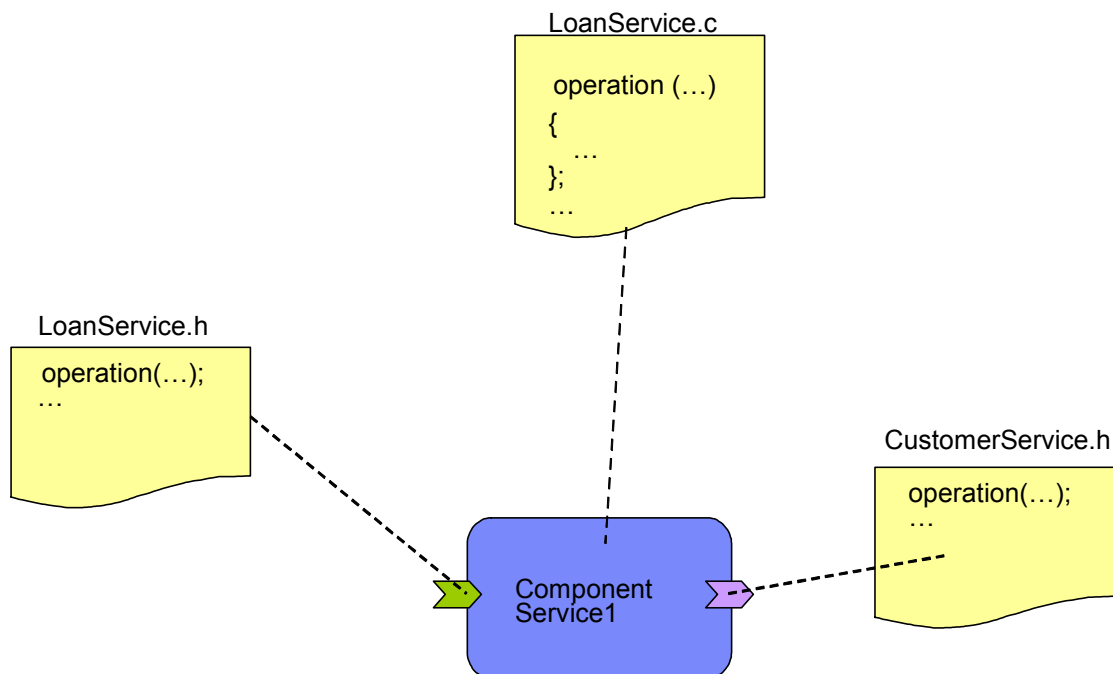
71

```
72 <?xml version="1.0" encoding="ASCII"?>  
73 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">  
74     <service name="LoanService">  
75         <interface.c header="LoanService.h"/>  
76     </service>  
77 </componentType>
```

78

79 The following picture shows the relationship between the C header files and implementation files for a  
80 component that has a single service and a single reference.





82

### 83 2.1.1 Implementing a Remotable Service

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

87

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

94

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

97

98 An implementation of a remotable service can declare whether it allows pass by reference data exchange  
 99 semantics on calls to it, meaning that the by-value semantics can be maintained without requiring that the  
 100 parameters be copied. The implementation of a remotable service that allows pass by reference MUST  
 101 NOT alter its input data during or after the invocation, and MUST NOT modify return data after invocation.  
 102 The *allowsPassByReference=true* attribute on the implementation.c element of a remotable service is  
 103 used to either declare that calls to the whole interface allows pass by reference. , this attribute can be  
 104 used on a specific function.

### 105 2.1.2 Implementing a Local Service

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

## 107 2.2 Conversational and Non-Conversational services

108 A *requires="conversational"* attribute on an `interface.c` element indicates that the service contract is  
109 conversational as described in the Assembly Specification.

110

111 A non-conversational service, the default when no annotation is specified, indicates that the service  
112 contract is stateless between requests. A conversational service indicates that requests to the service are  
113 correlated.

## 114 2.3 Component and Implementation Scopes

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

119

120 Scopes are specified using the **scope** attribute of the `implementation.c` element.

121

122 When a scope is not specified in an implementation file, the SCA runtime will interpret the implementation  
123 scope as **stateless**.

124

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

127

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

129

```
130 <component name="LoanService">  
131   <implementation.c module="loan" scope="composite"/>  
132 </component>
```

133

134 Certain scoped implementations potentially also specify **lifecycle functions** which are called when an  
135 implementation is instantiated or the scope is expired. An implementation is either instantiated eagerly  
136 when the scope is started (specified by `scope="COMPOSITE" eagerInit="true"`), or lazily when the first  
137 client request is received. Lazy instantiation is the default for all scopes. The C implementation uses the  
138 **init="true"** attribute of a implementation function element to denote the function to be called upon  
139 initialization and the **destroy="true"** attribute for the function to be called when the scope ends. Note that  
140 only functions with no arguments may be annotated as lifecycle callbacks. Not all implementation styles  
141 are suitable when lifecycle functions are needed. In particular, lifecycle functions are generally not  
142 applicable for program-based implementations (see Some methods of an implementation have  
143 operational characteristics that need to be identified. This is done using a method child element of  
144 `implementation.c`

145

146 The following snippet shows the `implementation.c` schema with the schema for a method child element:

147

```
148 <?xml version="1.0" encoding="ASCII"?>  
149 <!-- ImplementationFunction schema snippet -->  
150 <implementation.c xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"... >  
151   <function ... />  
152 </implementation.c>
```

155

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

157 • **name** : **NCName (1..1)** – name of the function being decorated.

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

159 exchange semantics.

160 • **init** : **boolean (0..1)** – indicates this function should be called to initialize the implementation.

161 • **destroy** : **boolean (0..1)** – indicates this function should be called to cleanup the implementation.

162 Implementing a Service with a Program).

163

164 For *stateless* scoped implementations, the SCA runtime MUST prevent concurrent execution of methods

165 on an instance of that implementation. However, *composite* scoped implementations MUST be able to

166 handle multiple threads running its methods concurrently.

167

168 The following sections specify the scopes an SCA runtime MUST support for C component

169 implementations.

### 170 **2.3.1 Stateless scope**

171 For stateless components, there is no implied correlation between service requests.

172 Lifecycle functions are not defined for stateless implementations.

### 173 **2.3.2 Request scope**

174 Service requests are delegated to the same implementation instance for all colocated service invocations

175 that occur while servicing a remote service request, i.e. a copy of the binary implementing the component

176 is loaded into a new memory space for each remote request and all corresponding local requests are

177 processed by this remote request-specific instance. The lifecycle of a request scope extends from the

178 point a request on a remotable interface enters the SCA runtime and a thread processes that request until

179 the thread completes synchronously processing the request.

180

181 There are times when a local request scoped service is called without a remotable service earlier in the

182 call stack, such as when a local service is called from a non-SCA entity. In these cases, a remote request

183 is always considered to be present, but the lifetime of the request is implementation dependent. For

184 example, a timer event could be treated as a remote request.

185

186 Request scope supports both `init="true"` and `destroy="true"` functions.

### 187 **2.3.3 Composite scope**

188 All service requests are dispatched to the same implementation instance for the lifetime of the containing

189 composite, i.e. the binary implementing the component is loaded into memory once and all requests are

190 processed by this single instance. The lifetime of the containing composite is defined as the time it

191 becomes active in the runtime to the time it is deactivated, either normally or abnormally.

192

193 Composite scope supports both `init="true"` and `destroy="true"` functions.

### 194 **2.3.4 Conversation scope**

195 A conversation is defined as a series of correlated interactions between a client and a target service. A

196 conversational scope starts when the first service request is dispatched to an implementation instance

197 offering the target service. A conversational scope completes after an end operation defined by the

198 service contract is called and completes processing or the conversation expires (see Operations that End

199 the Conversation). A conversation is potentially long-running and the SCA runtime MAY choose to

200 passivate implementation instances. If this occurs, the runtime MUST guarantee implementation instance  
201 state is preserved.

202

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

204

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

209

210 Lifecycle functions are not defined for conversational implementations.

## 211 **2.4 Implementing a Configuration Property**

212 Component implementations can be configured through properties. The properties and their types (not  
213 their values) are defined in the component type. The C component can retrieve properties values using  
214 the `SCAProperty<PropertyType>()` functions, for example `SCAPropertyInt()` to access an `Int` type  
215 property..

216

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

```
218 #include "SCA.h"  
219  
220 void clientFunction()  
221 {  
222     ...  
223     ...  
224     ...  
225     int32_t loanRating;  
226     int compCode, reason;  
227     ...  
228     ...  
229     ...  
230     SCAPropertyInt(L"maxLoanValue", &loanRating, &compCode, &reason);  
231     ...  
232     ...  
233     ...  
234 }
```

235

236 If the property is many valued, an array of the appropriate type is used as the second parameter, and the  
237 third parameter would point to an int that would receive the number of values. The type for the property  
238 SHOULD NOT allow more values to be defined than the size of the array in the implementation.

239

## 240 **2.5 Component Type and Component**

241 For a C component implementation, a component type is specified in a side file.

242

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

245

246 The following snippets show a C service interface and a C implementation of a service.

247

```
248 /* LoanService interface */
```

```
249     char approveLoan(long customerNumber, long loanAmount);
```

250

251 Implementation.

```
252     #include "LoanService.h"
253
254     char approveLoan(long customerNumber, long loanAmount)
255     {
256         ...
257     }
```

259

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

261

```
262     <?xml version="1.0" encoding="ASCII"?>
263     <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
264         <service name="LoanService">
265             <interface.c header="LoanService.h" componentType="LoanService" />
266         </service>
267     </componentType>
```

268

269 The following snippet shows the component using the implementation.

270

```
271     <?xml version="1.0" encoding="ASCII"?>
272     <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
273
274         name="LoanComposite" >
275
276         ...
277
278         <component name="LoanService">
279             <implementation.c module="loan"/>
280         </component>
281
282         ...
283
284     </composite>
```

## 285 2.5.1 Interface.c

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

288

```
289     <?xml version="1.0" encoding="ASCII"?>
290     <!-- interface.c schema snippet -->
291     <interface.c xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
292         header="NCName" remotable="boolean"? callbackHeader="Name"? >
293
294         <function ... />*
295         <callbackFunction ... />*
296
297     </interface.c>
```

298

299 The **interface.c** element has the following **attributes**:

- 300 • **header : Name (1..1)** – full name of the header file, including either a full path, or its equivalent, or a  
301 relative path from the composite root. This header file describes the interface.

- 302 • **callbackHeader : Name (0..1)** –full name of the header file that describes the callback interface,  
303 including either a full path, or its equivalent, or a relative path from the composite root.
- 304 • **remotable : boolean (0..1)** – indicates whether the service is remotable or local. The default is local.

305

306 The **interface.c** element has the following **child elements**:

- 307 • **function : CFunction (0..n)** – see Function and CallbackFunction
- 308 • **callbackFunction : CFunction (0..n)** – see Function and CallbackFunction

## 309 2.5.2 Function and CallbackFunction

310 Some functions of an interface have behavioral characteristics, which will be described later, that need to  
311 be identified. This is done using a function or callbackFunction child element of interface.cpp These  
312 child elements are also used when not all functions in a header file are part of the interface or when the  
313 interface is implemented by a program.

314

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

316

```
317 <?xml version="1.0" encoding="ASCII"?>
318 <!-- interface.c schema snippet -->
319 <interface.c xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"... >
320
321     <function name="NCName" oneWay="Boolean"? endsConversation="Boolean"?
322         input="NCName"? output="NCNAME"? /*>
323     <callbackFunction name="NCName" oneWay="Boolean"?
324         endsConversation="Boolean"? input="NCName"? output="NCName"? /*>
325
326 </interface.c>
```

327

328 The **function** and **callbackFunction** elements have the following **attributes**:

- 329 • **name : NCName (1..1)** – name of the function being decorated or included in the interface.
- 330 • **oneWay : boolean (0..1)** – see **Error! Reference source not found.**
- 331 • **endsConversation ; boolean (0..1)** – see Operations that End the Conversation
- 332 **input : NCNAME (0..1)** – If the interface is implemented by a program (seeSome methods of an  
333 implementation have operational characteristics that need to be identified. This is done using a method  
334 child element of implementation.c

335

336 The following snippet shows the implementation.c schema with the schema for a method child element:

337

```
338 <?xml version="1.0" encoding="ASCII"?>
339 <!-- ImplementationFunction schema snippet -->
340 <implementation.c xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"... >
341
342     <function ... /*>
343
344 </implementation.c>
```

345

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

- 347 • **name : NCName (1..1)** – name of the function being decorated.
- 348 • **allowsPassByReference : boolean" (0..1)** – indicates the method allows pass by reference data  
349 exchange semantics.

- 350 • **init : boolean (0..1)** – indicates this function should be called to initialize the implementation.
- 351 • **destroy : boolean (0..1)** – indicates this function should be called to cleanup the implementation.
- 352 • Implementing a Service with a Program), then this attribute identifies the `struct` defining the input
- 353 message format.
- 354 • **output : NCNAME (0..1)** – If the interface is implemented by a program, attribute identifies the
- 355 `struct` defining the output message format, if any.

### 356 2.5.3 Implementation.c

357 The following snippet shows the schema for the C implementation element used to define the  
358 implementation of a component.

```
359
360 <?xml version="1.0" encoding="ASCII"?>
361 <!-- implementation.c schema snippet -->
362 <implementation.c xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
363     module="NCName" library="boolean"? location="Name"
364     scope="scope"? componentType="Name" allowsPassByReference="Boolean"?
365     eagerInit="boolean"? init="boolean"? destroy="boolean"?
366     conversationMaxAge="string"? conversationMaxIdle="string"?
367     conversationSinglePrincipal="Boolean"? >
368
369     <function ... />*
370
371 </implementation.c>
```

372  
373 The **implementation.c** element has the following **attributes**:

- 374 • **module : NCName (1..1)** – name of the binary executable for the service component. This is the root
- 375 name of the module. A runtime specific prefix and/or suffix MAY be added.
- 376 • **library : boolean (0..1)** – indicates whether the service is implemented as a library or a program. The
- 377 default is library.
- 378 • **location : Name (0..1)** – location of the module as either a full path, its equivalent, or a relative path
- 379 from the composite root.
- 380 • **scope : CImplementationScope (0..1)** – indicates the scope of the component implementation. The
- 381 default is stateless.
- 382 • **componentType : Name (1..1)** – name of the componentType file. A “.componentType” extension
- 383 will be appended. An optional path to the componentType file which is relative to the root of the
- 384 composite MAY be included.
- 385 • **allowsPassByReference : boolean (0..1)** – indicates the service allows pass by reference data
- 386 exchange semantics on calls to it.
- 387 • **eagerInit : boolean (0..1)** – indicates a composite scoped implementation should be initialized
- 388 when it is loaded.
- 389 • **init : boolean (0..1)** – indicates program should be called with an initialize flag to initialize the
- 390 implementation.
- 391 • **destroy : boolean (0..1)** – indicates should be called with an destroy flag to to cleanup the
- 392 implementation.
- 393 • **conversationMaxAge : string (0..1)** – see Error! Reference source not found..
- 394 • **conversationMaxIdle : string (0..1)** – see Error! Reference source not found..
- 395 • **conversationSinglePrincipal : boolean (0..1)** – see Error! Reference source not found..

396  
397 The **interface.cpp** element has the following **child element**:



- 398 • **method : CImplementationMethod (0..n)** – see Implementation Function

## 399 2.5.4 Implementation Function

400 Some methods of an implementation have operational characteristics that need to be identified. This is  
401 done using a method child element of implementation.c

402

403 The following snippet shows the implementation.c schema with the schema for a method child element:

404

```
405 <?xml version="1.0" encoding="ASCII"?>  
406 <!-- ImplementationFunction schema snippet -->  
407 <implementation.c xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"... >  
408  
409     <function ... />*  
410  
411 </implementation.c>
```

412

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

- 414 • **name : NCName (1..1)** – name of the function being decorated.
- 415 • **allowsPassByReference : boolean" (0..1)** – indicates the method allows pass by reference data  
416 exchange semantics.
- 417 • **init : boolean (0..1)** – indicates this function should be called to initialize the implementation.
- 418 • **destroy : boolean (0..1)** – indicates this function should be called to cleanup the implementation.

## 419 2.6 Implementing a Service with a Program

420 Depending on the execution platform, services MAY be implemented in libraries, programs, or a  
421 combination of both libraries and programs. Services implemented as subroutines in a library are called  
422 directly by the runtime. Input and messages are passed as parameters, and output messages can either  
423 be additional parameters or a return value. Both local and remoteable interfaces are easily supported by  
424 this style of implementation.

425

426 For services implemented as programs, the SCA runtime uses normal platform functions to invoke the  
427 program. Accordingly, a service implemented as a program will run in its own address space and in its  
428 own process and its interface is most appropriately marked as remotable. A service implemented in a  
429 program will have either stateless or conversation scope. Local services implemented as subroutines  
430 used by a service implemented in a program can run in the address space and process of the program.

431

432 While it is possible to pass parameters to a C program, typically only a return code can be returned. To  
433 provide complete functionality for services implemented as programs, including allowing a program to  
434 implement multiple services, a runtime MAY provide a set of support functions.

435

436 Since a program can implement multiple services and often will implement multiple operations, the  
437 program has to query the runtime to determine which service and operation caused the program to be  
438 invoked. This is done using SCAService() and SCAOperation(). Once the specific service and operation  
439 is known, the proper input message can be retrieved using SCAMessageIn(). Once the logic of the  
440 operation is finished SCAMessageOut() is used to provide the return data to the runtime to be  
441 marshalled.

442

443 Since a program does not have a specific prototype for each operation of each service it implements, a C  
444 interface definition for the service MUST identify the operation names and the input and output message



445 formats using functions elements, with input and output attributes, in an interface.c element. Alternatively,  
446 an external interface definition, such as a WSDL document, is used to describe the operations and  
447 message formats.

448

449 The following shows a program implementing a service using these support functions.

450

```
451 #include "SCA.h"
452 #include "myInterface.h"
453 main () {
454     wchar_t myService [255];
455     wchar_t myOperation [255];
456     int compCode, reason;
457     struct FirstInputMsg myFirstIn;
458     struct FirstOutputMsg myFirstOut;
459
460
461     SCAService(myService, &compCode, &reason);
462
463     SCAOperation(myOperation, &compCode, &reason);
464
465     if (wstricmp(myOperation,L"myFirstOperation")==0){
466         SCAMessageIn(myService, myOperation,
467                     sizeof(struct FirstInputMsg), (void *)&myFirstIn,
468                     &compCode, &reason);
469         ...
470         SCAMessageOut(myService, myOperation,
471                      sizeof(struct FirstOutputMsg), (void *)&myFirstOut,
472                      &compCode, &reason);
473     }
474     else
475     {
476         ...
477     }
478 }
```

479

## 3 Basic Client Model

480 This section describes how to get access to SCA services from both SCA components and from non-SCA  
481 components. It also describes how to call operations of these services.

### 3.1 Accessing Services from Component Implementations

482 A service can get access to another service using a reference of the current component

484

485 The following shows the function used for this.

486

```
487 void SCALocate(wchar_t *referenceName, SCAREF *referenceToken,  
488               int *compCode, int *reason);  
489 void SCAInvoke(SCAREF referenceToken, wchar_t *operationName,  
490               int inputMsgLen, void *inputMsg,  
491               int outputMsgLen, void *outputMsg, int *compCode, int *reason);
```

492

493 The following shows a sample of how a service is called in a C component implementation.

494

```
495 #include "SCA.h"  
496  
497 void clientFunction()  
498 {  
499  
500     SCAREF serviceToken;  
501     int compCode, reason;  
502     long custNum = 1234;  
503     short rating;  
504  
505     ...  
506     SCALocate(L"customerService", &serviceToken, &compCode, &reason);  
507     SCAInvoke(serviceToken, L"getCreditRating", sizeof(custNum),  
508               (void *)&custNum, sizeof(rating), (void *)&rating,  
509               &compCode, &reason);  
510  
511 }
```

512

513 If a reference has multiple targets, the client has to use SCALocateMultiple() to retrieve tokens for each of  
514 the tokens and then invoke the operation(s) for each target. For example:

515

```
516 SCAREF *tokens;  
517 int num_targets;  
518 ...  
519 myFunction(...) {  
520     int compCode, reason;  
521     ...  
522     SCALocateMultiple(L"myReference", &tokens, &num_targets, &compCode,  
523                       &reason);  
524     for (i = 0; i < num_targets; i++)  
525     {  
526         // set up callback function  
527         SCASetCallback(tokens[i], L"myCallback", pfn, &compCode, &reason);  
528         // set up arguments  
529         SCAInvoke(tokens[i], L"myOperation", sizeof(inputMsg),
```

```
530         (void *)&inputMsg, 0, NULL, &compCode, &reason);
531     };
532 };
533
```

## 534 **3.2 Accessing Services from non-SCA component implementations**

535 Non-SCA components can access component services by obtaining an SCAREF from the SCA runtime  
536 and then following the same steps as a component implementation as described above.

537

538 How an SCA runtime implementation allows access to and returns a SCAREF is not defined by this  
539 specification.

## 540 **3.3 Calling Service Operations**

541 The previous sections show the various options for getting access to a service and using SCAInvoke to  
542 invoke operations of that service.

543

544 If you have access to a service whose interface is marked as remotable, then on calls to operations of  
545 that service you will experience remote semantics. Arguments and return values are passed **by-value**  
546 and it is possible to get a SCA\_SERVICE\_UNAVAILABLE reason code which is a Runtime error.

---

## 547 4 Conversational Services

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

### 552 4.1 Conversational Client

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

```
556  
557 #include "SCA.h"  
558 #include "LoanApplicationClient.h"  
559 #include "LoanService.h"  
560 #include "LoanApplication.h"  
561  
562 SCAREF serviceToken;  
563  
564 void clientFunction(struct LoanApplication loanApp, int term)  
565 {  
566  
567     long customerNumber = 1234;  
568     int compCode, reason;  
569  
570     SCALocate(L"loanService", &serviceToken, &compCode, &reason);  
571  
572     SCAInvoke(serviceToken, L"apply", sizeof(loanApp), (void *)&loanApp,  
573              0, NULL, &compCode, &reason);  
574     SCAInvoke(serviceToken, L"lockCurrentRate", sizeof(term), (void *)&term,  
575              0, NULL, &compCode, &reason);  
576  
577 }  
578  
579 bool isApproved()  
580 {  
581     int status;  
582     int compCode, reason;  
583  
584     SCAInvoke(serviceToken, L"getLoanStatus", 0, NULL,  
585              sizeof(status), (void *)&status, &compCode, &reason);  
586     return (status == 1);  
587 }  
588
```

### 588 4.2 Conversational Service Provider

589 A implementation which provides a service with a conversational interface can have any scope. In  
590 particular, it is not necessary for the implementation to have conversation scope. However, the provider of  
591 the conversational service is not required to write special code to maintain state if the implementation has  
592 conversation scope since the runtime maintains state associated with the conversation, by routing each  
593 operation invocation associated with the conversation to the same logical instance of the implementation,  
594 with state data held in instance variables. However, for implementations with other scopes, when an  
595 operation of a conversational interface is executing, the SCAGetConversationID() returns the  
596 conversation ID of the conversation. The conversation ID can be used by the implementation as an index  
597 to store and to look up state data associated with the conversation, using some suitable storage  
598 mechanism.

599

```
600     SCAGetConversationID(wchar_t *serviceName, wchar_t **ID, int *compCode,  
601                          int *reason);
```

602

603 The service implementation might also have optional conversation attributes control the lifetime and  
604 operation of the conversations it supports. These attributes are specified on the implementation.c  
605 element:

- 606 • **maxIdleTime** - The maximum time that can pass between operations within a single conversation. If  
607 more time than this passes, then the SCA runtime MAY end the conversation.
- 608 • **maxAge** - The maximum time that the entire conversation can remain active. If more time than this  
609 passes, then the SCA runtime MAY end the conversation.
- 610 • **singlePrincipal** – If true, only the principal (the user) that started the conversation has authority to  
611 continue the conversation.

612

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

615

616 Not specifying timeouts means that timeouts are defined by the implementation of the SCA runtime,  
617 however it chooses to do so.

618

619 Here is an example implementation definition of a conversational service.

620

```
621 <component name="LoanService">  
622     <implementation.c module="loan" conversationMaxAge="30"/>  
623 </component>
```

## 624 4.3 Operations that End the Conversation

625 An operation of a conversation scoped implementation can be marked with an endsConversation="true"  
626 attribute. This means that once this operation has been called, no further operation is to be called,  
627 allowing both the client and the target to free up resources that were associated with the conversation. It  
628 is also possible to mark an operation on a callback interface (described later) as  
629 endsConversation="true", in order for the service provider to be the party that chooses to end the  
630 conversation. If an operation is called after the conversation completes, the  
631 SCA\_CONVERSATION\_ENDED reason code is returned. This can also occur if there is a race condition  
632 between the client and the service provider calling their respective endsConversation operations. Calling  
633 the SCAEndConversation() function also ends a conversation.

634

635 The following is an example implementation with a function that has been annotated as ending a  
636 conversation:

637

```
638 <?xml version="1.0" encoding="ASCII"?>  
639 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">  
640     <service name="LoanService">  
641         <interface.c header="LoanService.h" remotable="true"  
642             requires="conversational">  
643             <function name="cancelApplication" endsConversation="true" />  
644         </interface.c>  
645     </service>  
646 </componentType>
```

647

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

## 649 **4.4 Conversation Lifetime Summary**

### 650 *Starting conversations*

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

- 652 • A service is located using SCALocate().

653

### 654 *Continuing conversations*

655 The client can continue an existing conversation, by:

656 Holding the service token that was created when the conversation started

657

### 658 *Ending conversations*

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

- 660 • A service operation marked endsConveration="true" is called.
- 661 • The service calls an operation marked endsConversation="true" on the callback interface.
- 662 • The service's conversation lifetime timeout occurs.
- 663 • The client calls SCAEndConversation().
- 664 • The client calls SCASetConversationID() which implicitly ends any active conversation.

665

666 If an operation is invoked after an operation marked endsConversation="ture" has been called then a new  
667 conversation will automatically be started. If SCAGetConversationID() is called after an operation  
668 marked endsConversation="true" is called, but before the next conversation has been started, it will return  
669 null.

670

671 If a service operation is called after the service provider's conversation timeout has caused the  
672 conversation to be ended, then the SCA\_CONVERSATION\_ENDED reason code will be returned. In  
673 order to use that service for a new conversation, the SCAEndConversation() operation has to be called.

## 674 **4.5 Application Specified Conversation IDs**

675 It is also possible to take advantage of the state management aspects of conversational services while  
676 using a client-provided conversation ID. To do this, the client would use SCASetConversationID().

677

```
678     wchar_t conversationID[] = L"myID";  
679     SCAREF serviceToken;  
680     int compCode, reason;  
681  
682     SCALocate(L"MyService", &serviceToken, &compCode, &reason);  
683     SCASetConversationID(serviceToken, conversationID, &compCode, &reason);
```

684

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

687

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

## 689 **4.6 Accessing Conversation IDs from Clients**

690 Whether the conversation ID is chosen by the client or is generated by the system, the client accesses  
691 the conversation ID of a conversation by calling the SCAGetConversationID().

692

693 If the conversation ID is not application specified, then SCAGetConversationID() is only guaranteed to  
694 return a valid value after the first operation has been invoked, otherwise it returns null.

---

## 695 5 Asynchronous Programming

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

### 704 5.1 Non-blocking Calls

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

708

709 Any function that returns "void" and only uses by-value parameters can be marked by using the  
710 **oneWay=true** attribute in the interface definition of the service. This means that the function is non-  
711 blocking and communication with the SCA runtime MAY use a binding that buffers the requests and  
712 sends it at some later time.

713

714 The following snippet shows the component type for a service with the reportEvent() function declared as  
715 a one-way operation:

716

```
717 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">  
718   <service name="LoanService">  
719     <interface.c header="LoanService.h">  
720       <function name="reportEvent" oneWay="true" />  
721     </interface.c>  
722   </service>  
723 </componentType>
```

724

725 SCA does not currently define a mechanism for making non-blocking calls to functions that return values.  
726 It is recommended that service designers define one-way operations as often as possible, in order to give  
727 the greatest degree of binding flexibility to deployers.

### 728 5.2 Callbacks

729 Callbacks services are used by *bidirectional services* as defined in the Assembly Specification  
730 **[ASSEMBLY]**:

731

732 A callback interface is declared by the *callbackHeader* and *callbackFunctions* attributes in the interface  
733 definition of the service. The following snippet shows the component type for a service *MyService* with the  
734 interface defined in *MyService.h* and the interface for callbacks defined in *MyServiceCallback.h*,

735

```
736 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" >  
737   <service name="MyService">  
738     <interface.c header="MyService.h" callbackHeader="MyServiceCallback.h"/>  
739   </service>  
740 </componentType>
```



## 741 5.2.1 Stateful Callbacks

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

744 A component gets access to the callback service by using the SCAGetCallback() function.

745

746 The following is an example service implementation for the service and callback declared above. When  
747 someFunction has completed its processing it retrieves the callback service and invokes a callback  
748 operation.

749

```
750 #include "SCA.h"  
751 #include "MyService.h"  
752 #include "MyServiceCallback.h"  
753  
754 void someFunction(int arg)  
755 {  
756     SCAREF serviceRef;  
757     int compCode, reason;  
758     ...  
759     /* do some processing... */  
760     SCAGetCallback(L"", &serviceRef, &compCode, &reason);  
761     SCACallback(serviceRef, L"receiveResult",  
762                 sizeof (result), (void *)&result, 0, NULL,  
763                 &compCode, &reason);  
764 }
```

765

766 The following shows how a client component would invoke the MyService service and receive the  
767 callback.

768

```
769 #include "SCA.h"  
770 #include "MyService.h"  
771 #include "MyServiceCallback.h"  
772  
773 void clientFunction(int arg)  
774 {  
775     SCAREF serviceToken;  
776     int compCode, reason;  
777  
778     /* invoke the service */  
779     SCALocate(L"MyService", &serviceToken, &compCode, &reason);  
780     SCAInvoke(serviceToken, L"someFunction", sizeof(arg), (void *)&arg, 0, NULL,  
781               &compCode, &reason);  
782 }  
783  
784 receiveResult(int result)  
785 {  
786     /* code to process result */  
787 }
```

788

789 Stateful callbacks do not require that any additional parameters be passed with service operations. This  
790 can be a great convenience. If the service has many operations and any of those operations could be the  
791 first operation of the conversation, it would be unwieldy to have to take a callback parameter as part of  
792 every operation, just in case it is the first operation of the conversation. It is also more natural than  
793 requiring the application developers to invoke an explicit operation whose only purpose is to pass the  
794 callback object to be used.

## 795 5.2.2 Stateless Callbacks

796 A stateless callback interface is a callback whose interface is not **conversational**. Unlike stateful  
797 services, the client that uses stateless callbacks will not have callback operations routed to an instance of  
798 the client that contains any state that is relevant to the conversation. As such, it is the responsibility of  
799 such a client to perform any persistent state management itself. The only information that the client has  
800 to work with (other than the parameters of the callback operation) is a callback ID that is passed with  
801 requests to the service and is guaranteed to be returned with any callback.

802

803 The following snippets show a client setting a callback id before invoking the asynchronous service and  
804 the callback function retrieving the callback ID:

```
805 SCAREF serviceToken;  
806 int compCode, reason;  
807  
808 void clientFunction(int arg)  
809 {  
810     /* invoke the service */  
811     SCALocate(L"MyService", &serviceToken, &comCode, &reason);  
812     SCASetCallbackID(serviceToken, L"1234" , &compCode, &reason);  
813     SCAInvoke(serviceToken, L"someFunction", sizeof(arg), (void *)&arg,  
814               0, NULL, &compCode, &reason);  
815 }  
816  
817  
818 receiveResult(int result)  
819 {  
820     wchar_t *myID;  
821     int compCode, reason;  
822  
823     SCAGetCallbackID(serviceToken, &myID, &compCode, &reason);  
824  
825     /* code to process result */  
826 }
```

## 827 5.2.3 Implementing Multiple Bidirectional Interfaces

828 Since it is possible for a single component to implement multiple services, it is also possible for callbacks  
829 to be defined for each of the services that it implements. The service name parameter of  
830 SCAGetCallback() identifies the service for which the callback is to be obtained.

## 831 5.2.4 Customizing the Callback Identity

832 The identity that is used to identify a callback request is, by default, generated by the SCA runtime.  
833 However, it is possible to provide an application specified identity to be used to identify the callback by  
834 using SCASetCallbackID(). This can be used even either stateful or stateless callbacks. The identity will  
835 be sent to the service provider, and the binding MUST guarantee that the SCA runtime will send the ID  
836 back when any callback operation is invoked.

837

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

---

## 842 6 Error Handling

843 Clients calling service operations will experience business logic errors, and SCA runtime errors.

844

845 Business logic errors are generated by the implementation of the called service operation. They are  
846 handled by client the invoking the operation of the service.

847

848 SCA runtime errors are generated by the SCA runtime and signal problems in the management of the  
849 execution of components, and in the interaction with remote services. The SCA C API includes two return  
850 codes on every function, a completion code and a reason code. The reason code is used to provide  
851 more detailed information if a function does not complete successfully. Currently the following SCA codes  
852 are defined:

853

```
854 /* Completion Codes */  
855 #define SCACC_OK 0  
856 #define SCACC_WARNING 1  
857 #define SCACC_FAULT 2  
858 #define SCACC_ERROR 3  
859  
860 /* Reason Codes */  
861 #define SCA_SERVICE_UNAVAILABLE 1  
862 #define SCA_CONVERSATION_ENDED 2  
863 #define SCA_NO_REGISTERED_CALLBACK 3  
864 #define SCA_MULTIPLE_SERVICES 4  
865 #define SCA_DATA_TRUNCATED 5
```

866

867 Reason codes between 0 and 100 are reserved for use by this specification. Vendor defined reason  
868 codes SHOULD start at 101.

869

## 7 C API

870

### 7.1 Synchronous Programming

871

The following shows the C interface declarations for synchronous programming.

872

873

```
typedef void *SCAREF;
```

874

875

```
void SCALocate(wchar_t *referenceName,
```

876

```
    SCAREF *referenceToken,
```

877

```
    int *compCode,
```

878

```
    int *reason);
```

879

880

```
void SCALocateMultiple(wchar_t *referenceName,
```

881

```
    SCAREF **referenceTokens,
```

882

```
    int *num_targets,
```

883

```
    int *CompCode,
```

884

```
    int *Reason);
```

885

886

```
void SCAInvoke(SCAREF referenceToken,
```

887

```
    wchar_t *operationName,
```

888

```
    int inputMsgLen,
```

889

```
    void *inputMsg,
```

890

```
    int outputMsgLen,
```

891

```
    void *outputMsg,
```

892

```
    int *compCode,
```

893

```
    int *reason);
```

894

895

```
void SCAPropertyBoolean(wchar_t *propertyName,
```

896

```
    char *value,
```

897

```
    int *compCode,
```

898

```
    int *reason);
```

899

900

```
void SCAPropertyByte(wchar_t *propertyName,
```

901

```
    int8_t *value,
```

902

```
    int *compCode,
```

903

```
    int *reason);
```

904

905

```
void SCAPropertyBytes(wchar_t *propertyName,
```

906

```
    int8_t **value,
```

907

```
    int *size,
```

908

```
    int *compCode,
```

909

```
    int *reason);
```

910

911

```
void SCAPropertyChar(wchar_t *propertyName,
```

912

```
    wchar_t *value,
```

913

```
    int *compCode,
```

914

```
    int *reason);
```

915

916

```
void SCAPropertyChars(wchar_t *propertyName,
```

917

```
    wchar_t **value,
```

918

```
    int *size,
```

919

```
    int *compCode,
```

920

```
    int *reason);
```

921

922

```
void SCAPropertyCChar(wchar_t *propertyName,
```

923

```
    char *value,
```

924

```
    int *compCode,
```

925

```
    int *reason);
```

```

926
927 void SCAPropertyCChars(wchar_t *propertyName,
928                       char **value,
929                       int *size,
930                       int *compCode,
931                       int *reason);
932
933 void SCAPropertyShort(wchar_t *propertyName,
934                      int16_t *value,
935                      int *compCode,
936                      int *reason);
937
938 void SCAPropertyInt(wchar_t *propertyName,
939                   int32_t *value,
940                   int *compCode,
941                   int *reason);
942
943 void SCAPropertyLong(wchar_t *propertyName,
944                    int64_t *value,
945                    int *compCode,
946                    int *reason);
947
948 void SCAPropertyFloat(wchar_t *propertyName,
949                      float *value,
950                      int *compCode,
951                      int *reason);
952
953 void SCAPropertyDouble(wchar_t *propertyName,
954                       double *value,
955                       int *compCode,
956                       int *reason);
957
958 void SCAPropertyString(wchar_t *propertyName,
959                      wchar_t **value,
960                      int *compCode,
961                      int *reason);
962
963 void SCAPropertyCString(wchar_t *propertyName,
964                        char **value,
965                        int *compCode,
966                        int *reason);
967
968 void SCAPropertyDate(wchar_t *propertyName,
969                    struct tm *value,
970                    int *compCode,
971                    int *reason);
972
973 void SCAGetFaultMessage(SCAREF referenceToken,
974                       int bufferLen,
975                       char *buffer,
976                       int *compCode,
977                       int *reason);
978
979 void SCASetFaultMessage(wchar_t *serviceName,
980                       wchar_t *operationName,
981                       int bufferLen,
982                       char *buffer,
983                       int *compCode,
984                       int *reason);

```

985

986 The C Synchronous Programming interface has the following functions:

- 987 • **SCALocate()** – provides a token that can be used to call an operation of a service. An error will be
- 988 returned if the reference resolves to more than one service.

- 989 • **SCALocateMultiple()** – provides a pointer to an array of tokens, one for each target of a reference  
990 with multiple targets.
- 991 • **SCAInvoke()** – invokes an operation of a service.
- 992 • **SCAProperty<PropertyType>()** – provides the value of the specified property. This API is available  
993 for Boolean, Byte, Bytes, Char, Chars, CChar, CChars, Short, Int, Long, Float, Double, String,  
994 CString, and Date. The Char, Chars, and String variants return wchar\_t based data while the CChar,  
995 CChars, and CString variants return char base data.  
996 The Bytes, Chars, and CChars variants return a buffer of data; a size parameter is used to specify the  
997 maximum size of the buffer and is updated to return the actual size of the data. The String and  
998 CString variants return a null terminated string; a size parameter is used to specify the maximum  
999 length of the string that can be returned. If the value for the parameter is too large to fit in the  
1000 provided space, CompCode is set to SCACC\_WARNING, Reason is set to  
1001 SCA\_DATA\_TRUNCATED, and the size parameter is set to the space required to hold the entire  
1002 value.
- 1003 • **SCAGetFaultMessage()** – retrieve a fault message for a service operation.
- 1004 • **SCASetFaultMessage()** – set a fault message for a service operation.

## 1005 7.2 Program-Based Implementation Support

1006 A runtime MAY additionally provide the following functions to support C implementations in programs:

1007

```

1008 void SCAService(wchar_t *serviceName, int *compCode, int *reason);
1009
1010 void SCAOperation(wchar_t *operationName, int *compCode, int *reason);
1011
1012 void SCAMessageIn(wchar_t *serviceName,
1013                  wchar_t *operationName,
1014                  int bufferLen,
1015                  void *buffer,
1016                  int *compCode,
1017                  int *reason);
1018
1019 void SCAMessageOut(wchar_t *serviceName,
1020                   wchar_t *operationName,
1021                   int bufferLen,
1022                   void *buffer,
1023                   int *CompCode,
1024                   int *Reason);

```

1025

1026 The C program-based implementation support has the following functions:

- 1027 • **SCAService()** – retrieve the name of the invoked service.
- 1028 • **SCAOperation()** – retrieve the name of the invoked service operation.
- 1029 • **SCAMessageIn()** – retrieve the input message for a service operation.
- 1030 • **SCAMessageOut()** – set the output message for a service operation.

## 1031 7.3 Asynchronous Programming

1032 The following shows the C interface declarations for asynchronous programming.

1033

```

1034 void SCAGetCallback(wchar_t *serviceName,
1035                   SCAREF *referenceToken,
1036                   int *compCode,
1037                   int *reason);
1038

```

```

1039 void SCACallback(SCAREF referenceToken,
1040                 wchar_t *operationName,
1041                 int inputMsgLen,
1042                 void *inputMsg,
1043                 int outputMsgLen,
1044                 void *outputMsg,
1045                 int *compCode,
1046                 int *reason);
1047
1048 void SCASetCallbackID(SCAREF referenceToken,
1049                      wchar_t *id,
1050                      int *compCode,
1051                      int *reason);
1052
1053 void SCAGetCallbackID(SCAREF referenceToken,
1054                      wchar_t **id,
1055                      int *compCode,
1056                      int *reason);

```

1057

1058 The C Asynchronous Programming interface has the following functions:

- 1059 • **SCAGetCallback()** – provides a token that can be used to call an operation of a callback. Used by a  
1060 service provider
- 1061 • **SCACallback()** – invoke a callback operation. Used by a service provider.
- 1062 • **SCASetCallbackID()** – set the callback ID for a service instance. Used by a service requestor.
- 1063 • **SCAGetCallbackID()** – returns the callback ID for a service instance. Used by a service provider or  
1064 requestor.

## 1065 7.4 Conversational Services

1066 The following shows the C interface declarations for conversations.

1067

```

1068 void SCAGetConversationID(wchar_t *serviceName,
1069                          wchar_t **id,
1070                          int *compCode,
1071                          int *reason);
1072
1073 void SCASetConversationID(SCAREF referenceToken,
1074                          wchar_t *id,
1075                          int *compCode,
1076                          int *reason);
1077
1078 void SCAEndConversation(SCAREF referenceToken,
1079                       int *compCode,
1080                       int *reason);

```

1081

1082 The C Conversational Services interface has the following functions:

- 1083 • **SCAGetConversationID()** – returns the conversation ID. Used by a service provider.
- 1084 • **SCASetConversationID()** – sets a user provided conversation ID. Used by a service requestor.
- 1085 • **SCAEndConversation()** – end a conversation. Used by a service requestor.

---

## 1086 8 C to WSDL Mapping

1087 This section describes a mapping from a C header file used to define an interface to a WSDL description  
1088 of that interface. The intent is for implementations of this proposal to be able to deploy a service based  
1089 only on a C header definition and for a WSDL definition of that service to be generated from the C, either  
1090 at deploy or run time.

1091  
1092 This mapping currently only deals with producing document/literal wrapped style services and WSDL from  
1093 C header files. Support for additional styles, if provided SHOULD be consistent with the mapping  
1094 specified in this specification. Support for additional styles is implementation dependent.

### 1095 8.1 Parameter and Return Type mappings

1096 This section details how types used as parameters or return types in C function prototypes get mapped to  
1097 XML schema elements in the generated WSDL.

#### 1098 8.1.1 Built-in type mappings

C types	Notes	XML Type
_Bool		xsd:boolean
char	signed 8-bit <sup>1</sup>	xsd:byte
unsigned char	unsigned 8-bit <sup>1</sup>	xsd:unsignedByte
short	signed 16-bit <sup>1</sup>	xsd:short
unsigned short	unsigned 16-bit <sup>1</sup>	xsd:unsignedShort
int	signed 16-bit <sup>1</sup>	xsd:short
unsigned int	unsigned 16-bit <sup>1</sup>	xsd:unsignedShort
long	signed 32-bit <sup>1</sup>	xsd:int
unsigned long	unsigned 32-bit <sup>1</sup>	xsd:unsignedInt
long long	signed 64-bit <sup>1</sup>	xsd:long
unsigned long long	unsigned 64-bit <sup>1</sup>	xsd:unsignedLong
float	32-bit floating point (IEEE-754-1985) <sup>1</sup>	xsd:float
double	64-bit floating point (IEEE-754-1985) <sup>1</sup>	xsd:double
long double	64-bit floating point (platform dependent, IEEE-754-1985) <sup>1</sup>	xsd:double

---

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



char* or char array	A null-terminated UTF-8 encoded string	xsd:string
wchar_t* or wchar_t array	A null-terminated UTF-16 or UTF-32 encoded string <sup>2</sup>	xsd:string
struct tm		xsd:dateTime

1099

1100 For example, a C function prototype defined in a header such as:

1101

```
1102 long myFunction(char *name, int id, double value);
```

1103

1104 would generate a schema like:

1105

```
1106 <xsd:element name="myFunction">
1107   <xsd:complexType>
1108     <xsd:sequence>
1109       <xsd:element name="name" type="xsd:string"/>
1110       <xsd:element name="id" type="xsd:short"/>
1111       <xsd:element name="value" type="xsd:double"/>
1112     </xsd:sequence>
1113   </xsd:complexType>
1114 </xsd:element>
1115
1116 <xsd:element name="myFunctionResponse">
1117   <xsd:complexType>
1118     <xsd:sequence>
1119       <xsd:element name="myFunctionResponseData" type="xsd:int"/>
1120     </xsd:sequence>
1121   </xsd:complexType>
1122 </xsd:element>
```

### 1123 8.1.2 Binary data mapping

1124 Binary data, such as data passed via non-null-terminated char \* or char arrays, is not supported in this  
 1125 mapping. char \* and char array parameters and return types are always mapped to xsd:string, and are  
 1126 null-terminated. This requirement also applies to wchar\_t\* and wchar\_t array parameters.

### 1127 8.1.3 Array mapping

1128 C arrays passed in or out of functions get mapped as normal elements but with multiplicity allowed via the  
 1129 minOccurs and maxOccurs facets. E.g. a function prototype such as:

1130

```
1131 long myFunction(char* name, int idList[], double value);
```

1132

1133 would generate a schema like:

1134

```
1135 <xsd:element name="myFunction">
```

---

<sup>2</sup> The encoding associated with a wchar\_t variable is determined by the size of the wchar\_t type. If wchar\_t is a 16-bit type, UTF-16 is used, otherwise UTF-32 is used.

```

1136 <xsd:complexType>
1137   <xsd:sequence>
1138     <xsd:element name="name" type="xsd:string"/>
1139     <xsd:element name="idList" type="xsd:short"
1140       minOccurs="0" maxOccurs="unbounded"/>
1141     <xsd:element name="value" type="xsd:double"/>
1142   </xsd:sequence>
1143 </xsd:complexType>
1144 </xsd:element>

```

## 1145 8.1.4 Multi-dimensional array mapping

1146 Multi-dimensional arrays will need converting into nested elements. E.g. a function prototype such as:

```

1147
1148 long myFunction(int multiIdArray[][4][2]);

```

1149  
1150 would generate a schema like:

```

1151
1152 <xsd:element name="myFunction">
1153   <xsd:complexType>
1154     <xsd:sequence>
1155       <xsd:element name="multiIdArray"
1156         minOccurs="0" maxOccurs="unbounded"/>
1157       <xsd:complexType>
1158         <xsd:sequence>
1159           <xsd:element name="multiIdArray"
1160             minOccurs="4" maxOccurs="4"/>
1161           <xsd:complexType>
1162             <xsd:sequence>
1163               <xsd:element name="multiIdArray" type="xsd:short"
1164                 minOccurs="2" maxOccurs="2" />
1165             </xsd:sequence>
1166           </xsd:complexType>
1167         </xsd:sequence>
1168       </xsd:complexType>
1169     </xsd:sequence>
1170   </xsd:complexType>
1171 </xsd:element>
1172 </xsd:sequence>
1173 </xsd:complexType>
</xsd:element>

```

## 1174 8.1.5 Pointer mapping

1175 A C function prototype that uses the 'pass-by-pointer' style, defining parameters that are either pointers,  
1176 is not meaningful when applied to web services, which rely on serialized data. A C function prototype that  
1177 uses pointers will be converted to a WSDL operation that is defined as if the parameters were 'pass-by-  
1178 value', with the web-service implementation framework responsible for creating the value, obtaining its  
1179 pointer and passing that to the implementation function.

1180 E.g. a C function prototype defined in a header such as:

```

1181
1182 long myFunction(char *name, int *id, double *value);

```

1183  
1184 would generate a schema like:

```

1186 <xsd:element name="myFunction">
1187   <xsd:complexType>
1188     <xsd:sequence>
1189       <xsd:element name="name" type="xsd:string"/>
1190       <xsd:element name="id" type="xsd:short"/>
1191       <xsd:element name="value" type="xsd:double"/>
1192     </xsd:sequence>
1193   </xsd:complexType>
1194 </xsd:element>

```

1195

1196 Note here how the `char*` type is a special case – `char*` parameters map to `xsd:string`.

1197 Pointers are also used where in/out parameters are required – where the function changes the value of  
1198 the parameter and those changes are subsequently available in the invoking code – see In/Out  
1199 Parameters below.

## 1200 8.1.6 Struct mapping

1201 C structs that contain types that can be mapped, are themselves mapped to complex types. For  
1202 example, a function such as:

1203

```
1204 char *myFunction(struct DataStruct data, int id);
```

1205

1206 with the `DataStruct` type defined as a struct holding mappable types:

1207

```
1208 struct DataStruct {
1209     char *name;
1210     double value;
1211 };

```

1212

1213 would convert to a schema like:

1214

```

1215 <xsd:element name="myFunction">
1216   <xsd:complexType>
1217     <xsd:sequence>
1218       <xsd:element name="data" type="DataStruct" />
1219       <xsd:element name="id" type="xsd:int"/>
1220     </xsd:sequence>
1221   </xsd:complexType>
1222 </xsd:element>
1223
1224 <xsd:complexType name="DataStruct">
1225   <xsd:sequence>
1226     <xsd:element name="name" type="xsd:string"/>
1227     <xsd:element name="value" type="xsd:double"/>
1228   </xsd:sequence>
1229 </xsd:complexType>

```

1230

1231 `char` and `wchar_t` arrays inside of structs are not mapped to a restricted subtype of `xsd:string` that  
1232 limits the length the space allowed in the array.

1233

```
1234 struct DataStruct {
1235     char name[256];

```

```
1236     double value;
1237 };
1238
```

1239 would convert to a schema like:

```
1240
1241 <xsd:element name="myFunction">
1242   <xsd:complexType>
1243     <xsd:sequence>
1244       <xsd:element name="data" type="DataStruct" />
1245       <xsd:element name="id" type="xsd:int"/>
1246     </xsd:sequence>
1247   </xsd:complexType>
1248 </xsd:element>
1249
1250 <xsd:complexType name="DataStruct">
1251   <xsd:sequence>
1252     <xsd:element name="name">
1253       <xsd:simpleType>
1254         <xsd:restriction base="xsd:string">
1255           <xsd:maxLength value="255"/>
1256         </xsd:restriction>
1257       </xsd:simpleType>
1258     </xsd:element>
1259     <xsd:element name="value" type="xsd:double"/>
1260   </xsd:sequence>
1261 </xsd:complexType>
```

## 1262 8.1.7 Enum mapping

1263 In C `enums` define a list of named symbols that map to values. If a function uses an `enum` type, this can  
1264 be mapped to a restricted element in the WSDL schema.

1265 For example, a function such as:

```
1266
1267 char *getValueFromType(enum ParameterType type);
```

1268  
1269 with the `ParameterType` type defined as an `enum`:

```
1270
1271 enum ParameterType {
1272     UNSET = 1,
1273     TYPEA,
1274     TYPEB,
1275     TYPEC
1276 };
1277
```

1278 would convert to a schema like:

```
1279
1280 <xsd:element name="getValueFromType">
1281   <xsd:complexType>
1282     <xsd:sequence>
1283       <xsd:element name="type" type="ParameterType"/>
1284     </xsd:sequence>
1285   </xsd:complexType>
1286 </xsd:element>
```

```
1287
1288 <xsd:simpleType name="ParameterType">
1289   <xsd:restriction base="xsd:int">
1290     <xs:minInclusive value="1"/>
1291     <xs:maxInclusive value="4"/>
1292   </xsd:restriction>
1293 </xsd:simpleType>
```

1294  
1295 The restriction used will have to be appropriate to the values of the enum elements. For example, a non-  
1296 contiguous enum like:

```
1297
1298 enum ParameterType {
1299     UNSET = 'u',
1300     TYPEA = 'A',
1301     TYPEB = 'B',
1302     TYPEC = 'C'
1303 };
```

1304  
1305 would convert to a schema like:

```
1306
1307 <xsd:simpleType name="ParameterType">
1308   <xsd:restriction base="xsd:int">
1309     <xsd:enumeration value="86"/> <!-- Character 'u' -->
1310     <xsd:enumeration value="65"/> <!-- Character 'A' -->
1311     <xsd:enumeration value="66"/> <!-- Character 'B' -->
1312     <xsd:enumeration value="67"/> <!-- Character 'C' -->
1313   </xsd:restriction>
1314 </xsd:simpleType>
```

## 1315 8.1.8 Union mapping

1316 In C unions allow the same memory location to be used for different variables. Handling of C unions is not  
1317 defined by this mapping, and is implementation dependent. For portability it is recommended that unions  
1318 not be used in service interfaces.

## 1319 8.1.9 Typedef mapping

1320 Typedef mappings are supported by this specification, and will be followed when evaluating parameter  
1321 and return types. This mapping does not define whether typedef names will be used in the resulting  
1322 WSDL file. The use of these names is implementation dependent.

## 1323 8.1.10 Pre-processor mapping

1324 C allows for the use of pre-processor directives in order to control how a C header is parsed. Handling  
1325 for pre-processor directives is not defined by this mapping, and support is implementation dependent.  
1326 For portability it is recommended that pre-processor directives not be used in service interfaces.

## 1327 8.1.11 Nesting types

1328 If a `struct` or `enum` nests other `structs` or `enums`, it is mapped, as long as the nesting eventually  
1329 resolves to a mappable type. For example, a function such as:

```
1330
1331 char *myFunction(struct DataStruct data);
```

1332

1333 with types defined as follows:

1334

```
1335 struct DataStruct {
1336     char name[30];
1337     double values[20];
1338     ParameterType type;
1339 };
1340
1341 enum ParameterType {
1342     UNSET = 1,
1343     TYPEA,
1344     TYPEB,
1345     TYPEC
1346 };
```

1347

1348 would convert to a schema like:

1349

```
1350 <xsd:element name="myFunction">
1351     <xsd:complexType>
1352         <xsd:sequence>
1353             <xsd:element name="data" type="DataStruct"/>
1354         </xsd:sequence>
1355     </xsd:complexType>
1356 </xsd:element>
1357
1358 <xsd:complexType name="DataStruct">
1359     <xsd:sequence>
1360         <xsd:element name="name">
1361             <xsd:simpleType>
1362                 <xsd:restriction base="xsd:string">
1363                     <xsd:maxLength value="29"/>
1364                 </xsd:restriction>
1365             </xsd:simpleType>
1366         </xsd:element>
1367         <xsd:element name="values" type="xsd:double" minOccurs=20
1368 maxOccurs=20/>
1369         <xsd:element name="type" type=" ParameterType"/>
1370     </xsd:sequence>
1371 </xsd:complexType>
1372
1373 <xsd:simpleType name="ParameterType">
1374     <xsd:restriction base="xsd:int">
1375         <xs:minInclusive value="1"/>
1376         <xs:maxInclusive value="4"/>
1377     </xsd:restriction>
1378 </xsd:simpleType>
```

### 1379 **8.1.12 SDO mapping**

1380 C function prototypes that use dynamic SDO DATAOBJECT objects as parameter or return types are  
1381 mapped to the any type in the WSDL schema as the schema for a Data Object is unknown before  
1382 runtime. For example, a C function prototype defined in a header such as:

1383

```
1384 long myFunction(DATAOBJECT data);
```

1385

1386 would generate a schema like:

1387

```
1388 <xsd:element name="myFunction">
1389   <xsd:complexType>
1390     <xsd:sequence>
1391       <xsd:element name="data">
1392         <xsd:complexType>
1393           <xsd:sequence>
1394             <xsd:any processContents="skip"/>
1395           </xsd:sequence>
1396         </xsd:complexType>
1397       </xsd:element>
1398     </xsd:sequence>
1399   </xsd:complexType>
1400 </xsd:element>
```

1401

1402 Typed (static) Data Objects are supported via the rules for WSDL to C mapping below.

### 1403 **8.1.13 void \* mapping**

1404 The `void *` type is not supported due to its undefined nature.

### 1405 **8.1.14 Included types**

1406 An implementation MUST completely map a C interface to WSDL, including types (structs, enums, etc.)  
1407 that might be defined in a different C header than the one that is being mapped. Implementations  
1408 SHOULD allow a list of "include" directories to be specified. Types that are included (via a `#include`  
1409 `"SomeHeader.h"` statement) MUST be mappable to a schema element via the rules in this document.

## 1410 **8.2 Header mapping**

1411 Unless otherwise specified with annotations, all of the functions in a C header map to a single WSDL  
1412 service element, a single WSDL binding element and a single WSDL portType element. The WSDL  
1413 service element contains a single WSDL port element. The WSDL binding and WSDL portType elements  
1414 each contain multiple WSDL operation elements that map to the functions defined in the C header. A pair  
1415 of WSDL message elements and a pair of XML schema elements are generated for each WSDL  
1416 operation. SOAP binding and address information is also generated. For example, a C header  
1417 (MyService.h) containing:

1418

```
1419 int myFunction(char *data);
1420 double myOtherFunction(double otherData);
```

1421

1422 would generate WSDL like:

1423

```
1424 <?xml version="1.0" encoding="UTF-8"?>
1425 <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
1426   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
1427   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
1428   xmlns:tns="http://MyService"
1429   targetNamespace="http://MyService">
1430   <types>
1431     <xsd:schema targetNamespace="http://MyService"
1432       xmlns:xsd="http://www.w3.org/2001/XMLSchema"
1433       xmlns:tns="http://MyService">
```

```

1434         elementFormDefault="qualified">
1435
1436     <xsd:element name="myFunction">
1437         <xsd:complexType>
1438             <xsd:sequence>
1439                 <xsd:element name="data" type="xsd:string"/>
1440             </xsd:complexType>
1441         </xsd:element>
1442     <xsd:element name="myFunctionResponse">
1443         <xsd:complexType>
1444             <xsd:sequence>
1445                 <xsd:element name="myFunctionResponseData"
1446                     type="xsd:short"/>
1447             </xsd:complexType>
1448         </xsd:element>
1449
1450     <xsd:element name="myOtherFunction">
1451         <xsd:complexType>
1452             <xsd:sequence>
1453                 <xsd:element name="otherData" type="xsd:double"/>
1454             </xsd:sequence>
1455         </xsd:complexType>
1456     </xsd:element>
1457     <xsd:element name="myOtherFunctionResponse">
1458         <xsd:complexType>
1459             <xsd:sequence>
1460                 <xsd:element name="myOtherFunctionResponseData"
1461                     type="xsd:double"/>
1462             </xsd:complexType>
1463         </xsd:element>
1464
1465 </xsd:schema>
1466 </types>
1467
1468 <message name="myFunctionRequestMsg">
1469     <part name="body" element="tns:myFunction"/>
1470 </message>
1471 <message name="myFunctionResponseMsg">
1472     <part name="body" element="tns:myFunctionResponse"/>
1473 </message>
1474
1475 <message name="myOtherFunctionRequestMsg">
1476     <part name="body" element="tns:myOtherFunction"/>
1477 </message>
1478 <message name="myOtherFunctionResponseMsg">
1479     <part name="body" element="tns:myOtherFunctionResponse"/>
1480 </message>
1481
1482 <portType name="MyServicePortType">
1483     <operation name="myFunction">
1484         <input message="tns:myFunctionRequestMsg"/>
1485         <output message="tns:myFunctionResponseMsg"/>
1486     </operation>
1487     <operation name="myOtherFunction">
1488         <input message="tns:myOtherFunctionRequestMsg"/>
1489         <output message="tns:myOtherFunctionResponseMsg"/>
1490     </operation>
1491 </portType>

```



```

1492
1493 <binding name="MyServiceBinding" type="tns:MyService">
1494   <soap:binding style="document"
1495     transport="http://schemas.xmlsoap.org/soap/http"/>
1496   <operation name="myFunction">
1497     <soap:operation soapAction="MyService#myFunction"/>
1498     <input>
1499       <soap:body use="literal"/>
1500     </input>
1501     <output>
1502       <soap:body use="literal"/>
1503     </output>
1504   </operation>
1505   <operation name="myOtherFunction">
1506     <soap:operation soapAction="MyService#myOtherFunction"/>
1507     <input>
1508       <soap:body use="literal"/>
1509     </input>
1510     <output>
1511       <soap:body use="literal"/>
1512     </output>
1513   </operation>
1514 </binding>
1515
1516 <service name="MyService">
1517   <port name="MyServicePort" binding="tns:MyServiceBinding">
1518     <soap:address location="http://server:9090/MyService"/>
1519   </port>
1520 </service>
1521 </definitions>

```

1522

1523 If annotations are used to define multiple interfaces in a single C header file, then each interface maps to  
 1524 a unique Binding and PortType element and the service element contains multiple port elements.

## 1525 8.3 Function mapping

### 1526 8.3.1 Unnamed parameters

1527 Above, we have seen function prototypes with named parameters. C allows function declarations without  
 1528 parameter names, simply types. Functions defined in this way are not supported by this specification.

### 1529 8.3.2 The return type

1530 The return type in C functions is unnamed, so, as has been shown above, an implementation MUST  
 1531 generate a name for the elements required by doc-lit-wrapped WSDL. E.g. for the function prototype  
 1532 above, the response data will be returned using the following schema:

1533

```

1534 <xsd:element name="myFunctionResponse">
1535   <xsd:complexType>
1536     <xsd:sequence>
1537       <xsd:element name="myFunctionResponseData" type="xsd:short"/>
1538     </xsd:sequence>
1539   </xsd:complexType>
1540 </xsd:element>

```

### 1541 8.3.3 The void return type

1542 Handling of the void return type is controlled by the oneWay annotation. If oneWay is true, the operation  
1543 will be mapped to a one-way (in-only) WSDL operation, otherwise it will be mapped to a request-response  
1544 WSDL operation where the output message is empty.

1545

```
1546 void myFunction(char *name, double value);
```

1547

1548 would generate a schema like:

1549

```
1550 <xsd:element name="myFunctionRequestMsg">  
1551   <xsd:complexType>  
1552     <xsd:sequence>  
1553       <xsd:element name="name" type="xsd:string"/>  
1554       <xsd:element name="value" type="xsd:double"/>  
1555     </xsd:sequence>  
1556   </xsd:complexType>  
1557 </xsd:element>  
1558  
1559 <xsd:element name="myFunctionResponseMsg">  
1560   <xsd:complexType/>  
1561 </xsd:element>
```

1562

1563 and a WSDL operation in the WSDL portType and binding elements such as:

1564

```
1565 <portType name="MyServicePortType">  
1566   <operation name="myFunction">  
1567     <input message="tns:myFunctionRequestMsg"/>  
1568     <output message="tns:myFunctionResponseMsg"/>  
1569   </operation>  
1570 </portType>  
1571  
1572 <binding name="MyServiceBinding" type="tns:MyService">  
1573   <soap:binding style="document"  
1574     transport="http://schemas.xmlsoap.org/soap/http"/>  
1575   <operation name="myFunction">  
1576     <soap:operation soapAction="MyService#myFunction"/>  
1577     <input>  
1578       <soap:body use="literal"/>  
1579     </input>  
1580     <output>  
1581       <soap:body use="literal"/>  
1582     </output>  
1583   </operation>  
1584 </binding>
```

1585

1586 Alternatively, if the oneWay annotation is specified on the function:

1587

```
1588 /* @oneWay */  
1589 void myFunction(char* name, double value);
```

1590

1591 the following schema would be generated:

1592

```
1593 <xsd:element name="myFunctionRequestMsg">
1594   <xsd:complexType>
1595     <xsd:sequence>
1596       <xsd:element name="name" type="xsd:string"/>
1597       <xsd:element name="value" type="xsd:double"/>
1598     </xsd:sequence>
1599   </xsd:complexType>
1600 </xsd:element>
```

1601

1602 and a WSDL operation in the WSDL portType and binding elements that contains no output element,  
1603 such as:

1604

```
1605 <portType name="MyServicePortType">
1606   <operation name="myFunction">
1607     <input message="tns:myFunctionRequestMsg"/>
1608   </operation>
1609 </portType>
1610
1611 <binding name="MyServiceBinding" type="tns:MyService">
1612   <soap:binding style="document"
1613     transport="http://schemas.xmlsoap.org/soap/http"/>
1614   <operation name="myFunction">
1615     <soap:operation soapAction="MyService#myFunction"/>
1616     <input>
1617       <soap:body use="literal"/>
1618     </input>
1619   </operation>
1620 </binding>
```

### 1621 8.3.4 No Parameters Specified

1622 If a C function prototype has no parameters, the input schema element is still required (for doc-lit-  
1623 wrapped WSDL) but is empty. E.g. a function prototype:

1624

```
1625 int getValue();
```

1626

1627 would generate a schema like:

1628

```
1629 <xsd:element name="getValue">
1630   <xsd:complexType/>
1631 </xsd:element>
1632
1633 <xsd:element name="getValueResponse">
1634   <xsd:complexType>
1635     <xsd:sequence>
1636       <xsd:element name="getValueResponseData" type="xsd:short"/>
1637     </xsd:sequence>
1638   </xsd:complexType>
1639 </xsd:element>
```

1640

## 9 WSDL to C Mapping

1641 This section describes mapping from a WSDL description to a C header file that declares a set of  
1642 `structs`, `enums`, and functions that implement the interface. The intent of this proposal is for tools to be  
1643 able to generate a C header file based on the WSDL description.

1644 A WSDL file consists of a `/definitions` XML element which contains both logical and physical definitions of  
1645 a service contract. The logical contract is defined by the following XML elements in the WSDL file:

- 1646 • `/definitions/portType`
- 1647 • `/definitions/operation`
- 1648 • `/definitions/message`
- 1649 • `/definitions/types`

1650 The physical contract is defined by these XML elements:

- 1651 • `/definitions/service`
- 1652 • `/definitions/binding`
- 1653 • `/definitions/port`

1654 This specification assumes and recommends that any implemented system treat the physical elements of  
1655 the contract as runtime configured options, thus requiring either the WSDL file itself or some other derived  
1656 configuration file to be available at runtime. It is therefore assumed that no C code is generated from  
1657 these elements, and will not form part of a WSDL to C mapping specification.

1658 This mapping currently only deals with consuming document/literal wrapped style services.

### 9.1 Type Mapping

1660 The `<types>` element of a WSDL document encloses data type definitions that are relevant for the  
1661 exchanged messages. It is noted that any XML-based system for defining types would be acceptable in  
1662 this context. However, for maximum interoperability and platform neutrality, WSDL prefers the use of XML  
1663 Schema (XSD - <http://www.w3.org/2001/XMLSchema>) as the canonical type system, and treats it as the  
1664 intrinsic type system. XSD is therefore adopted for the purpose of these C specifications.

1665 This section describes how types are XSD types mapped to C.

#### 9.1.1 Simple type mapping

XML Type	C Mapping	Notes
<code>xsd:boolean</code>	<code>char</code>	
<code>xsd:byte</code>	<code>char</code>	signed 8-bit
<code>xsd:unsignedByte</code>	<code>unsigned char</code>	unsigned 8-bit
<code>xsd:short</code>	<code>short</code>	signed 16-bit
<code>xsd:unsignedShort</code>	<code>unsigned short</code>	unsigned 16-bit
<code>xsd:int</code>	<code>long</code>	signed 32-bit
<code>xsd:unsignedInt</code>	<code>unsigned long</code>	unsigned 32-bit
<code>xsd:long</code>	<code>long long</code>	signed 64-bit
<code>xsd:unsignedLong</code>	<code>unsigned long long</code>	unsigned 64-bit

xsd:float	float	32-bit floating point (IEEE-754-1985)
xsd:double	double	64-bit floating point (IEEE-754-1985)
xsd:string	wchar_t * or wchar_t[]	Array is used if the length is bounded
xsd:normalizedString	wchar_t * or wchar_t[]	Array is used if the length is bounded
xsd:token	wchar_t * or wchar_t[]	Array is used if the length is bounded
xsd:language	char * or char[]	Array is used if the length is bounded
xsd:Name	wchar_t * or wchar_t[]	Array is used if the length is bounded
xsd:NCName	wchar_t * or wchar_t[]	Array is used if the length is bounded
xsd:ID	wchar_t * or wchar_t[]	Array is used if the length is bounded
xsd:IDREF	wchar_t * or wchar_t[]	Array is used if the length is bounded
xsd:NMTOKEN	wchar_t * or wchar_t[]	Array is used if the length is bounded
xsd:NMTOKENS		Treat as list
xsd:QName	wchar_t * or wchar_t[]	Array is used if the length is bounded
xsd:dateTime	struct tm	
xsd:date	struct tm	
xsd:time	struct tm	
xsd:gDay	struct tm	
xsd:gMonth	struct tm	
xsd:gMonthDay	struct tm	
xsd:gYear	struct tm	
xsd:gYearMonth	struct tm	
xsd:duration	char *	
xsd:decimal	char *	double or float might be possible
xsd:integer	char *	long long, long, or short might be possible
xsd:positiveInteger	char *	unsigned long long, unsigned long, or unsigned short might be possible
xsd:negaitveInteger	char *	long long, long, or short might be possible
xsd:nonPositiveInteger	char *	long long, long, or short might be possible
xsd:nonNegativeInteger	char *	unsigned long long, unsigned long, or unsigned short might be possible
xsd:base64Binary	unsigned char * or unsigned char[]	Array is used if the length is bounded
xsd:hexBinary	char * or char[]	Array is used if the length is bounded

1668 **Notes:**

- 1669 • In general, `xsd:string` and types derived from `xsd:string` have to map to a combination of a  
1670 `wchar_t *` and a separately allocated data array. If either the `length` or `maxLength` facet is used,  
1671 then a `wchar_t []` is used. If the `pattern` facet is used, this might allow the use of `char` and/or also  
1672 constrain the length. Exploitation of the `pattern` facet in the mapping is optional.

1673

1674 **Example:**

```
1675 <xsd:element name="myString" type="xsd:string"/>
```

1676 maps to:

```
1677 wchar_t *myString;  
1678 /* this points to a dynamically allocated buffer with the data */  
1679  
1680 <xsd:simpleType name="boundedString25">  
1681 <xsd:restriction base="xsd:string">  
1682 <xsd:length value="25"/>  
1683 </xsd:restriction>  
1684 </xsd:simpleType>  
1685 ...  
1686 <xsd:element name="myString" type="boundedString25"/>  
1687 maps to:  
1688 wchar_t myString[26];
```

1689

- 1690 • Unbounded binary data is mapped to a `char *` that points to the location where the actual data is  
1691 located. Like strings if the binary data is bounded in length, a `char []` is used.

1692

1693 **Examples:**

```
1694 <xsd:element name="myData" type="xsd:hexBinary"/>
```

1695 maps to:

```
1696 char *myData;  
1697 /* this points to a dynamically allocated buffer with the data */  
1698  
1699 <xsd:simpleType name="boundedData25">  
1700 <xsd:restriction base="xsd:hexBinary">  
1701 <xsd:length value="25"/>  
1702 </xsd:restriction>  
1703 </xsd:simpleType>  
1704 ...  
1705 <xsd:element name="myData" type="boundedData25"/>
```

1706 maps to:

```
1707 char myData[26];
```

1708

- 1709 • In general, `xsd:decimal`, `xsd:integer` and the subsets of `xsd:integer` are mapped to  
1710 strings. It is possible that the value could be converted to a `double` or `long long` (or one of the  
1711 shorter numeric formats), but this is not guaranteed. If the `totalDigits`, or a bounding combination  
1712 of `minInclusive`, `minExclusive`, `maxInclusive`, and `maxExclusive` facets are used it might  
1713 be possible to guarantee mapping to one of the numeric formats.

1714

1715 **Examples:**

```
1716 <xsd:element name="myInteger" type="xsd:integer"/>
```

1717 maps to:

```
1718 char *myInteger;  
1719 /* this points to a dynamically allocated buffer with the data */
```

1720

```
1721 <xsd:element name="myBoundedInteger">  
1722   <xsd:simpleType>  
1723     <xsd:restriction base="xsd:integer">  
1724       <xsd:minInclusive value="-20000"/>  
1725       <xsd:maxInclusive value="20000"/>  
1726     </xsd:restriction>  
1727   </xsd:simpleType>  
1728 </xsd:element>
```

1729 maps to:

```
1730 long myBoundedInteger;
```

1731

```
1732 <xsd:element name="myDecimal" type="xsd:decimal"/>
```

1733 maps to:

```
1734 char *myDecimal;  
1735 /* this points to a dynamically allocated buffer with the data */
```

1736

```
1737 <xsd:element name="myBoundedDecimal">  
1738   <xsd:simpleType>  
1739     <xsd:restriction base="xsd:decimal">  
1740       <xsd:totalDigits value="10"/>  
1741       <xsd:fractionDigits value="2"/>  
1742     </xsd:restriction>  
1743   </xsd:simpleType>  
1744 </xsd:element>
```

1745 maps to:

```
1746 double myBoundedDecimal;
```

1747

- 1748 • Since C does not have a way of representing unset values, elements with `minOccurs !=`  
1749 `maxOccurs` and lists with `minLength != maxLength`, which have a variable but bounded number  
1750 of instances is variable, but bounded, are mapped to a count of the number of occurrences and an  
1751 array. If the count is 0, then the contents of the array is undefined. Lists are treated in the same  
1752 manner.

1753

#### 1754 Examples:

```
1755 <xsd:element name="dates" type="xsd:date" maxOccurs="5"/>
```

1756 maps to:

```
1757 size_t dates_num;  
1758 struct tm date[5];
```

1759

```
1760 <xsd:simpleType name="lineNumList">  
1761   <xsd:list itemType="xsd:int"/>  
1762 </xsd:simpleType>  
1763 <xsd:simpleType name="lineNumList6">  
1764   <xsd:restriction base="lineNumList">  
1765     <xsd:minLength value="1"/>  
1766     <xsd:maxLength value="6"/>  
1767   </xsd:restriction>
```

```
1768 </xsd:simpletype>
1769 ...
1770 <xsd:element name="lineNums" type="lineNumList6"/>
```

1771 maps to:

```
1772 size_t lineNums_num;
1773 long lineNums[6];
```

- 1774
- 1775 • Since C does not allow for unbounded arrays, elements with `maxOccurs = unbounded` and lists  
1776 without a defined `length` or `maxLength`, are mapped to a count of the number of occurrences and  
1777 a pointer to the location where the actual data is located as an array

1778

1779 **Examples:**

```
1780 <xsd:element name="dates" type="xsd:date" maxOccurs="unbounded"/>
```

1781 maps to:

```
1782 size_t dates_num;
1783 struct tm *date;
1784 /* this points to a dynamically allocated array of struct tm's */
```

```
1785
1786 <xsd:simpleType name="lineNumList">
1787   <xsd:list itemType="xsd:int"/>
1788 </xsd:simpleType>
1789 ...
1790 <xsd:element name="lineNums" type="lineNumList"/>
```

1791 maps to:

```
1792 size_t lineNums_num;
1793 long *lineNums;
1794 /* this points to a dynamically allocated array of longs */
```

- 1795
- 1796 • An enumeration facet on numeric types MAY be mapped to an enum that contains the valid values to  
1797 aid input validation. An enumerations facet on types derived from string MAY be mapped to a static  
1798 array of strings that represent the valid values to aid input validation.
  - 1799 • Union Types are not supported.

## 1800 9.1.2 Complex type and group mapping

1801 Complex types and groups are mapped to `structs` with the attributes and elements of the `type` mapped  
1802 to members of the `struct`.

- 1803 • The name of the `struct` is the name of the `type` or `group`.
- 1804 • Attributes are placed at the top of the `struct`.
- 1805 • Simple types are mapped to members as described above.
- 1806 • The same rules for variable number of instances of a simple type element apply to complex type  
1807 elements.
- 1808 • A `sequence` group is mapped as either a simple type or a complex type as appropriate.

1809

1810 **Example:**

```
1811 <xsd:complexType name="myType">
1812   <xsd:sequence>
1813     <xsd:element name="name">
1814       <xsd:simpleType>
```



```

1815     <xsd:restriction base="xsd:string">
1816         <xsd:length value="25"/>
1817     </xsd:restriction>
1818 </xsd:simpleType>
1819 </xsd:element>
1820 <xsd:element name="idList" type="xsd:int"
1821             minOccurs="0" maxOccurs="unbounded"/>
1822 <xsd:element name="value" type="xsd:double"/>
1823 </xsd:sequence>
1824 </xsd:complexType>

```

1825 maps to:

```

1826 struct myType {
1827     wchar_t name[26];
1828     size_t idList_num;
1829     long *idList;
1830     /* this points to a dynamically allocated array of longs */
1831     double value;
1832 };

```

1833

- 1834 • While XML Schema allow the elements of an all group to appear in any order, the order is fixed in
- 1835 the C mapping. An all group is mapped as pointer to a struct and an instance of the struct. If the
- 1836 group is not present, the pointer is NULL and the values of the members of the struct are undefined.

1837

1838 **Example:**

```

1839 <xsd:element name="myVariable">
1840     <xsd:complexType name="myType">
1841         <xsd:all>
1842             <xsd:element name="name">
1843                 <xsd:simpleType>
1844                     <xsd:restriction base="xsd:string">
1845                         <xsd:length value="25"/>
1846                     </xsd:restriction>
1847                 </xsd:simpleType>
1848             </xsd:element>
1849             <xsd:element name="idList" type="xsd:int"
1850                 minOccurs="0" maxOccurs="unbounded"/>
1851             <xsd:element name="value" type="xsd:double"/>
1852         </xsd:all>
1853     </xsd:complexType>
1854 </xsd:element>

```

1855 maps to:

```

1856 struct myType {
1857     wchar_t name[26];
1858     size_t idList_num;
1859     long *idList;
1860     /* this points to a dynamically allocated array of longs */
1861     double value;
1862 } *pmyVariable, myVariable;

```

1863

- 1864 • Choice groups are not supported.
- 1865 • Nillable elements are mapped to a pointer to the value and the value itself. If the element is not
- 1866 present, the pointer is NULL and the value is undefined.

1867

1868 **Example:**

```

1869 <xsd:element name="shipDate" type="xsd:date" nillable="true"/>

```

1870 maps to:

```
1871 struct tm *pshipDate, shipDate;
```

1872

- 1873 • Mixed content is not supported.
- 1874 • Open content (Any Attribute and Any Element) is not supported.

## 1875 9.2 Message Mapping

1876 A `<message>` element can be thought of as an abstract collection of data types to be used in the context  
1877 of a C function for sending data to or from a server. The `<part>` elements can be considered to be  
1878 parameters in a C function, its type determining the C type of that parameter. A message containing no  
1879 parts is thus equivalent to a function containing no parameters.

## 1880 9.3 Part Mapping

1881 See the next section for details on the C mapping for this element, since the context of the containing  
1882 `<message>` element within an `<operation>` determines the exact C mapping.

1883 **Note:** Although WSDL 1.1 schema does not require either the `element` or `type` attributes, these two  
1884 attributes are required on a mutually exclusive basis to successfully map to C.

## 1885 9.4 Operation Mapping

1886 The `<operation>` element is used to specify a set of abstract operations, usually using the abstract  
1887 messages also defined in the WSDL document. This element will be one of two types, described in  
1888 Sections 9.4.2 One-way Operation and 9.4.3 Request-Response Operation. In addition, a name,  
1889 `/definitions/portType/operation[@name]` is required.

1890 For each occurrence of an operation element in the WSDL document an implementation **MUST** create a  
1891 corresponding function in the header file corresponds to the WSDL `<portType>` element which contains  
1892 the operation. The parts within the input and output messages themselves are passed as parameters  
1893 to this function, while fault messages, where present, are returned via a separate API.

1894 Zero or more `<message>` elements are used within an `<operation>` element to specify a collection of  
1895 operations for WSDL services, and which also determine the “direction” of the messages (*in* or *out*). See  
1896 Section 9.4.1.1 for details of the C mapping of operations.

1897 This function is named `/definitions/portType/operation[@name]`. If this name starts or ends in  
1898 the string literal “`Operation`”, this will be stripped, provided that the remaining string forms a valid and  
1899 unique C identifier.

### 1900 9.4.1 Operation Elements `<input>`, `<output>` and `<fault>`

1901 Operations fall into one of two WSDL groups:

- 1902 • `wsdl:request-response-or-one-way-operation`, used for messages from a client to a  
1903 server. The operation consists of a required `<input>` element which references a message from the  
1904 client to the server, and an optional `<output>` element which refers to a response message from  
1905 the server. If the `<output>` element is present, then optional `<fault>` elements might also be  
1906 present, which are used to raise exception messages during the processing of the operation by the  
1907 server.
- 1908 • `wsdl:solicit-response-or-notification-operation`, used for messages from a server to  
1909 a client. This specification does not require support for `wsdl:solicit-response-or-`  
1910 `notification-operation`.

1911 These elements in turn, through use of `<input>`, `<output>`, and `<fault>` elements, use two WSDL  
1912 types:

- 1913 • wsdl:tParam, used for <input> and <output> message types, and which requires a message attribute  
1914 (/definitions/portType/operation/input[@message] and /definitions/portType/  
1915 operation/output[@message]) to be specified. An optional name attribute  
1916 (/definitions/portType/operation/input[@name] and /definitions/portType/operation/output[@name])  
1917 might be specified. The message attribute contributes to the parameter list of the operation through  
1918 its parts, while the name attribute if present is ignored.
  - 1919 • wsdl:tFault, used for the <fault> message type, which require both a message (/definitions/  
1920 portType/operation/fault[@message]) and name (/definitions/portType/operation/ fault[@name])  
1921 attribute to be specified.
- 1922 The details of the C mapping are discussed in the next section.

### 1923 9.4.1.1 C Mapping of <operation> Elements

1924 The implementation MUST map each <operation> element to a single C function in the file  
1925 corresponding to the <portType> element in which it is contained. The sum of the <part> elements  
1926 within the <message> elements which are referenced within each <operation> element maps to the  
1927 parameter list of the corresponding C function. The rules for these mappings are detailed below.

### 1928 Function Mapping Rules

1929 The rules for mapping function are as follows:

- 1930 • Each <operation> element maps to a single C function. However, if the WSDL definition (including  
1931 the name attribute) of two such operations inside the same <portType> element are identical  
1932 (which is legal in WSDL 1.1), then the implementaqtion MUST generate a second C function with an  
1933 extended or mangled name.
- 1934 • All functions return type void.
- 1935 • The function name is mapped from /definitions/portType/operation[@name].
- 1936 • All values and references are passed through the C operation parameter list.

### 1937 Parameter Types

1938 The following terminology, similar to that of IDL, is introduced:

- 1939 • A parameter that sends data from the client to the server only is referred to as an **In** parameter.
- 1940 • A parameter that sends data from the server to the client only is referred to as an **Out** parameter.
- 1941 • A parameter that sends data in both directions between the client and the server is referred to as an  
1942 **InOut** parameter.

1943 It follows therefore, that all one-way operations have only *In* parameters and similarly, all notification  
1944 operations have only *Out* parameters. For request-response operations, parts in messages that are used  
1945 either within <input> or <output> elements in any given operation are *In* and *Out* parameters  
1946 respectively. *InOut* parameters, however, arise in two ways:

- 1947 • Where a message is common to both <input> and <output> elements within the same  
1948 <operation>, each of the <part> elements within that message map to a single *InOut* parameter.

1949

### 1950 Example:

```
1951 <!-- WSDL -->
1952 ...
1953 <message name="StockIDMessage">
1954   <part name="exchangeSymbol" type="xs:string">
1955   <part name="stockSymbol" type="xs:string">
1956 </message>
1957 ...
1958 <portType ...>
1959   <operation name="GetNextStockSymbol">
```

```

1960     <input message="StockIDMessage">
1961     <output message="StockIDMessage">
1962     </operation>
1963     ...
1964 </portType>
1965     ...

```

1966 In this example, both `exchangeSymbol` and `stockSymbol` map as *InOut* parameters.

- 1967 • Where two or more `<part>` elements, each in different `<message>` elements, share the same  
1968 name and type attributes, and these messages are used independently in `<input>` and `<output>`  
1969 elements, then that `<part>` element maps to a single *InOut* parameter.

1970

### 1971 Example:

```

1972 <!-- WSDL -->
1973 ...
1974 <message name="StockIDMessage">
1975   <part name="exchangeSymbol" type="xs:string">
1976   <part name="stockSymbol" type="xs:string">
1977 </message>
1978 <message name="StockQuoteMessage">
1979   <part name="stockSymbol" type="xs:string">
1980   <part name="high" type="xs:decimal">
1981   <part name="low" type="xs:decimal">
1982   <part name="last" type="xs:decimal">
1983 </message>
1984 ...
1985 <portType ...>
1986   <operation name="GetStockPrice">
1987     <input message="StockIDMessage">
1988     <output message="StockQuoteMessage">
1989   </operation>
1990   ...
1991 </portType>
1992     ...

```

1993 In this example, `stockSymbol` maps as a single *InOut* parameter because `<part>` elements of the  
1994 same name and type are used in the context of both input and output.

## 1995 Parameter Mapping Rules

1996 The rules for mapping function parameters are as follows:

- 1997 • Each `<part>` element within each of the `<message>` elements referenced in the `<operation>`  
1998 elements map as a parameter of the corresponding operation. The parameter type is the C mapping  
1999 of one of either the XML type `/definitions/message/part[@type]` or the XML type of the  
2000 element `/definitions/message/part[@element]` of the referenced part.
- 2001 • The parameter name is mapped from `/definitions/message/part[@name]`.
- 2002 • All generated C parameter names are unique. Where two or more `<part>` elements to be mapped  
2003 into a C function have a common name attribute but have differing type attributes (from multiple  
2004 `<message>` elements within an `<operation>`), each of these is mapped to the parameter list with  
2005 distinct names. This is achieved by appending from the string sequence `"_1"`, `"_2"`, etc. to the part  
2006 name so that the final parameter name is unique.
- 2007 • Where two or more `<part>` elements to be mapped into a C function have both common name *and*  
2008 type attributes (from multiple `<message>` elements within a single `<operation>`), the element is  
2009 assumed to refer to a common data element, and is mapped once only to the parameter list. It is not  
2010 necessary to change the name in any way unless the name is also associated with other types as  
2011 described in the previous rule.

2012 For reasons of code portability, the order of the parameters in the function is significant. To accommodate  
2013 this, the following rules are mandated:

- 2014 • Where the `parameterOrder` attribute  
2015 `/definitions/portType/operation[@parameterOrder]` is present, the order of the  
2016 supplied list determines the parameter order of the C operation, provided that there is a complete  
2017 correlation between the value of the `parameterOrder` attribute and the parameters for that  
2018 operation as passed in the WSDL source file. It is mandated that an exception be thrown if there is a  
2019 mismatch between the `parameterOrder` attribute value and the actual operation parameters.
- 2020 • In the event that no `parameterOrder` attribute exists, then the order is determined first by the parts  
2021 of the `<input>` element (if present), then by those of the `<output>` element (if present),  
2022 irrespective of the order in which these elements appear within the `<operation>` element. In the  
2023 event that any *InOut* parameters are encountered, the first appearance of that parameter using the  
2024 above rules will stand as the parameter position in the parameter list, the second and subsequent  
2025 locations where that parameter would otherwise appear are ignored.

2026

### 2027 Example:

```
2028 <!-- WSDL -->  
2029 ...  
2030 <message name="StockIDMessage">  
2031 <part name="exchangeSymbol" type="xs:string">  
2032 <part name="stockSymbol" type="xs:string">  
2033 </message>  
2034 <message name="StockQuoteMessage">  
2035 <part name="stockSymbol" type="xs:string">  
2036 <part name="high" type="xs:float">  
2037 <part name="low" type="xs:float">  
2038 <part name="last" type="xs:float">  
2039 </message>  
2040 ...  
2041 <portType ...>  
2042 <operation name="GetStockPrice">  
2043 <input message="StockIDMessage">  
2044 <output message="StockQuoteMessage">  
2045 </operation>  
2046 ...  
2047 </portType>  
2048 ...
```

2049 results in the C function

```
2050 void GetStockPrice(wchar_t *exchangeSymbol,  
2051                  wchar_t *stockSymbol,  
2052                  float *high,  
2053                  float *low,  
2054                  float *last);
```

## 2055 Fault Mapping

2056 It is possible for WSDL request-response operations to define and raise exceptions through the `<fault>`  
2057 element. Such faults are mapped as C `structs`.

2058 The implementation **MUST** generate a `struct` that exposes the C mapping of the message `<part>s`.  
2059 These follow the mapping rules for simple and complex XML types outlined in this specification.

2060 A `<message>` element referenced by a `<fault>` element is referred to as a **fault message** throughout  
2061 this section. Note, however, that a WSDL fault message might also be used in the context of `<input>`  
2062 and/or `<output>` elements, and might thus not be used exclusively in the context of a fault.

2063 The C mapping rules for user faults are as follows:

- 2064 – The `struct` is derived from the `<message>` `<part>s`.

- 2065 • There is a 1:1 relationship between the `<message>` element to which the `<fault>` element refers  
2066 (the *fault message*) and the C struct. Thus, two faults (each with unique names in different  
2067 operations) and which refer to the same *fault message* will make use of the same C struct.
- 2068 • The struct name is the string `/definitions/message[@name]` of the *fault message*. If this string  
2069 ends in the literal "Exception", then the string is used as-is; however if this is not the case, then  
2070 the string literal "Exception" is added to the name. For example, fault message `<message`  
2071 `name="StockIDMessage">` would map to a C struct named "StockIDMessageException".
- 2072 • The required fault name attribute `/definitions/portType/operation/fault[@name]` is the  
2073 first member of the struct.
- 2074 • The data of each part element within the *fault message* is stored within a member of the struct  
2075 following the mapping rules defined above.

2076

2077 **Example:**

2078 For a `<part>` named *partName* of an XML type that maps to a built-in C type *builtInType* within a  
2079 message named *FaultException* the mapping is as follows:

```
2080 /* C */
2081 struct FaultException {
2082     ...
2083     builtInType partName;
2084     ...
2085 };
```

2086 For a `<part>` named *partName* of an XML type that maps to a non-built-in C *type* within a message  
2087 named *FaultException*:

```
2088 /* C */
2089 struct FaultException {
2090     ...
2091     struct type partName;
2092     ...
2093 };
```

2094 **9.4.2 One-way Operation**

2095 The function is invoked by the client, but no response is expected from or sent by the server. This  
2096 operation type contains a single `<input>` element. Only System Exceptions are allowed in one-way mode;  
2097 User Exceptions will not be thrown back to the client.

2098

2099 **Example:**

```
2100 <!-- WSDL -->
2101 ...
2102 <message name="ReadyMessage">
2103     <part name="boolFlag" type="xs:boolean"/>
2104 </message>
2105 ...
2106 <portType name="StockQuoteEngine">
2107     <operation name="StartEngine">
2108         <input message="tns:ReadyMessage"/>
2109     </operation>
2110 </portType>
2111 ...
```

2112 maps to:

```
2113 /* C */
2114 ...
```

```
2115 void StartEngine(char boolFlag);
2116 ...
```

### 2117 9.4.3 Request-Response Operation

2118 The client invokes the function, and waits for the server to send a response. This operation type contains  
2119 a single `<input>` element followed by a single `<output>` element. This might be followed by the  
2120 optional `<fault>` element.

2121

#### 2122 Example:

```
2123 <!-- WSDL -->
2124 ...
2125 <message name="VoidMessage"/>
2126 <message name="ReadyMessage">
2127   <part name="boolFlag" type="xs:boolean"/>
2128 </message>
2129 ...
2130 <portType name="StockQuoteEngine">
2131   <operation name="IsEngineReady">
2132     <input message="tns:VoidMessage"/>
2133     <output message="tns:ReadyMessage"/>
2134   </operation>
2135 </portType>
2136 ...
```

2137 maps to:

```
2138 /* C */
2139 ...
2140 void IsEngineReady(char *boolFlag);
2141 ...
```

### 2142 9.5 PortType Mapping

2143 For each occurrence in the WSDL file of `/definitions/portType`, a unique header file SHOULD be  
2144 created. Optionally all of the occurrences MAY be placed in a single header file with `@SERVICE`  
2145 annotations to separate the occurrences. This file contains function declarations corresponding to the  
2146 operations defined within this element.

2147

#### 2148 Example:

```
2149 <!-- WSDL -->
2150 ...
2151 <portType name="TutorialA">
2152   ...
2153 </portType>
2154 <portType name="TutorialBExample">
2155   ...
2156 </portType>
2157 ...
```

2158 results in C mapping:

```
2159 TutorialA.h
2160 /* ... operations ... */
2161 and
2162 TutorialBExample.h
2163 /* ... operations ... */
```

2164 or

```
2165 <wsdl>.h
2166 /* @Service(name="TutorialA") */
2167 /* ... operations ... */
2168
2169 /* @Service(name="TutorialBExample") */
2170 /* ... operations ... */
```

## 2171 9.6 Name Mapping

2172 For each WSDL or XML Schema element name that is mapped to a C name, the following conventions  
2173 are adopted:

- 2174 • If the string contains a hyphen, or any other special character not allowed in an identifier, convert it  
2175 into an underscore.
- 2176 • If the string is a keyword then append an underscore to it.
- 2177 • If the string starts with a digit, or any other character that is not allowed as an initial character of an  
2178 identifier, then prepend an underscore to it.
- 2179 • Duplicate names in the same scope are made unique by the addition of one or two numeric digits to  
2180 the second and subsequent instances of the name.
- 2181 • For example, three instances of `year` become `year`, `year1` and `year2`.



---

## 2182 10Packaging

### 2183 10.1 Composite Packaging

2184 The physical realization of an SCA composite is a folder in a file system containing at least one  
2185 .composite file. The following shows the MyValueComposite just after creation in a file system.

2186

```
2187     MyValue/  
2188         MyValue.composite
```

2189

2190 Besides the .composite file the composite contains artifacts that define the implementations of  
2191 components, and that define the bindings of services and references. Examples of artifacts would C  
2192 header files, shared libraries (for example, dll), WSDL portType definitions, XML schemas, WSDL binding  
2193 definitions, and so on. These artifacts can be contained in subfolders of the composite, whereby  
2194 programmers have the freedom to construct a folder structure that best fits the needs of their project. The  
2195 following shows the complete MyValue composite folder file structure in a file system.

2196

```
2197     MyValue/  
2198         MyValue.composite  
  
2199         bin/  
2200             myvalue.dll  
2201         services/myValue/  
2202             MyValue.h  
2203             MyValue.componentType  
2204             MyValueService.wsdl  
2205         services/customer/  
2206             CustomerService.h  
2207             Customer.h  
2208         services/stockquote/  
2209             StockQuoteService.h  
2210             StockQuoteService.wsdl
```

2212

2213 Note that the folder structure is not architected, other than the .composite file MUST be at the root of the  
2214 folder structure.

2215

2216 **Addressing of the resources** inside of the composite is done relative to the root of the composite (i.e.  
2217 the location of the .composite file).

2218 Shared libraries (including dlls) will be located as specified in the <implementation.c> element in the  
2219 .composite file relative to the root of the composite.

2220 XML definitions like XML schema complex types or WSDL portTypes are referenced by composite and  
2221 component type files using URI's. These URI's consist of the namespace and local name of these XML  
2222 definitions. The composite folder or one of its subfolders has to contain the XML resources providing the  
2223 XML definitions identified by these URI's.

---

## 2224 **11 Types Supported in Service Interfaces**

2225 A service interface can support a restricted set of the types available to a C programmer. This section  
2226 summarizes the valid types that can be used.

### 2227 **11.1 Local service**

2228 For a local service the types that are supported are:

- 2229 • Any of the C primitive types (for example, int, short, char). In this case the types will be passed by  
2230 value as is normal for C.
- 2231 • Pointers to any of the C primitive types (for example, int \*, short \*, char \*).
- 2232 • DATAOBJECT. An SDO handle.

### 2233 **11.2 Remotable service**

2234 For a remotable service being called by another service the data exchange semantics is by-value. In this  
2235 case the types that are supported are:

- 2236 • Any of the C primitive types (for example, int, short, char). This will be copied.
- 2237 • DATAOBJECT. An SDO handle. The SDO will be copied and passed to the destination.

2238 Unless the interface is marked as allowing pass by reference semantics, the behavior of the following are  
2239 not defined:

- 2240 • Pointers.

---

2241 **12 Restrictions on C header files**

2242 A C header file that is used to describe an interface has some restrictions. It MUST:

- 2243
- Declare at least one function

2244

2245 The following C keywords and constructs MUST NOT be used:

- 2246
- Macros



2248

## A C Annotations

2249 This section provides definitions of the annotations which can be used in the interface and  
2250 implementation headers. The annotations are defined as C comments in interface header and  
2251 implementation files, for example:

2252

```
2253 /* @Scope("stateless") */
```

2254

2255 All meta-data that is represented by annotations can also be defined in .composite and .componentType  
2256 side files as defined in the SCA Assembly Specification and the extensions defined in this specification.  
2257 Component type information found in the component type file must MUST be compatible with any  
2258 specified annotation information.

### 2259 A.1 Application of Annotations to C Program Elements

2260 In general an annotation immediately precedes the program element it applies to. If multiple annotations  
2261 apply to a program element, all of the annotations SHOULD be in the same comment block.

- 2262 • Function or Function Prototype

2263 The annotation immediately precedes the function definition or declaration.

2264 Example:

```
2265 /* @OneWay */  
2266 reportEvent(int eventID);
```

- 2267 • Variable

2268 The annotation immediately precedes the variable definition.

2269 Example:

```
2270 /* @Property */  
2271 long loanType;
```

- 2272 • Set of Functions Implementing a Service

2273 A set of functions implementing a service begins with an @Service annotations. Any  
2274 annotations applying to this service as a whole immediately precede the @Service  
2275 annotation. These annotations SHOULD be in the same comment block as the @Service  
2276 annotation.

2277 Example

```
2278 /* @Scope("composite")  
2279 * @Service(name="LoanService", interfaceHeader="loan.h") */
```

- 2280 • Set of Function Prototypes Defining an Interface

2281 To avoid any ambiguity about the application of an annotation to a specific function or  
2282 the set of functions defining an interface, if an annotation is to apply to the interface as  
2283 a whole, then the @Interface annotation must be used, even in the case where there is  
2284 just one interface defined in a header file. Any annotations applying to the interface  
2285 immediately precede the @Interface annotation. These annotations SHOULD be in the  
2286 same comment block as the @Interface annotation.

```
2287 /* @Conversational  
2288 * @Interface(name="LoanService" */
```

## 2289 A.2 Interface Header Annotations

2290 This section lists the annotations that may be used in the header file that defines a service interface.

### 2291 A.2.1 @Interface

2292 Annotation that indicates the start of a new interface definition.

2293

2294 **Corresponds to:** interface.c element

2295

2296 **Format:**

```
2297 /* @Interface (name="serviceName") */
```

2298 where

- 2299 • **name : NCName (1..1)** – specifies the name of the service.

2300

2301 **Applies to:** Set of functions defining an interface.

2302 Function declarations following this annotation form the definition of this interface. This annotation also  
2303 serves to bound the scope of the remaining annotations in this section,

2304

2305 Example:

2306 Implementation:

```
2307 /* @Interface (name="LoanService") */
```

2308

2309 Service definition:

```
2310 <service name="LoanService">  
2311   <interface.c header="loans.h" />  
2312 </service>
```

### 2313 A.2.2 @Operation

2314 Annotation that indicates that a function defines an operation of a service. There are two formats for this  
2315 annotation depending on if the service is implemented as a set of subroutines or in a program.

2316

2317 **Corresponds to:** function element of an interface.c element

2318

2319 If the service is implemented as a set of subroutines, this format is used.

2320

2321 **Format:**

```
2322 /* @Operation (name="operationName") */
```

2323 where

- 2324 • **name : NCName (0..1)** – gives the operation a different name than the function name.

2325

2326 **Applies to (library based implementations):** Function declaration

2327 The function declaration following this annotation defines an operation of the current service. If no  
2328 @Operation annotation exists in an interface definition, all the function declarations in a header file or  
2329 following an @Interface annotation define the operations of a service, otherwise only the annotated  
2330 function declarations define operations for the service.

2331

2332 Example:

2333 Interface header (loans.h):

```
2334 short internalFcn(char *param1, short param2);
2335
2336 /* @Operation(name="getRate") */
2337 void rateFcn(char *cust, float *rate);
```

2338

2339 Interface definition:

```
2340 <interface.c header="loans.h">
2341     <operation name="getRate" />
2342 </interface.c>
```

2343

2344 If the service is implemented in a program, the following format is used. In this format, all operations must  
2345 be defined via annotations.

2346

2347 **Format:**

```
2348 /* @Operation(name="operationName", input="inputStruct", output="outputStruct") */
2349 */
```

2350 where

- 2351 • **name: NCName (1..1)** – specifies the name of the operation.
- 2352 • **input : NCName (1..1)** – specifies the name of a struct that defines the format of the input message.
- 2353 • **output : NCName (0..1)** – specifies the name of a struct that defined the format of the output  
2354 message if one is used.

2355

2356 **Applies to (program based implementations):** struct declarations

2357

2358 Example

2359 Interface header (loans.h):

```
2360 /* @Operation(name="getRate", input="rateInput", output="rateOutput") */
2361 struct rateInput {
2362     char cust[25];
2363     int term;
2364 };
2365 struct rateOutput {
2366     float rate;
2367     int rateClass;
2368 };
```

2369

2370 Interface definition:

```
2371 <interface.c header="loans.h">
2372     <operation name="getRate" input="rateInput" output="rateOutput"/>
2373 </interface.c>
```

## 2374 A.2.3 @Remotable

2375 Annotation on service interface to indicate that a service is remotable.

2376

2377 **Corresponds to:** remotable="true" attribute of interface.c element.

2378

2379 **Format:**

2380 `/* @Remotable */`

2381 The default is **false** (not remotable).

2382

2383 **Applies to:** Interface

2384

2385 Example

2386 Interface header (LoanService.h):

2387 `/* @Remotable */`

2388

2389 Service definition:

```
2390 <service name="LoanService">  
2391   <interface.c header="LoanService.h" remotable="true" />  
2392 </service>
```

## 2393 **A.2.4 @Callback**

2394 Annotation on a service interface to specify the callback interface.

2395

2396 **Corresponds to:** callbackHeader attribute of interface.c element.

2397

2398 **Format:**

2399 `/* @Callback(header="headerName") */`

2400 where

- 2401 • **header : Name (1..1)** – specifies the name of the header defining the callback service interface.

2402

2403 **Applies to:** Interface

2404

2405 Example

2406 Interface header (MyService.h):

2407 `/* @Callback(header="MyServiceCallback.h") */`

2408

2409 Service definition:

```
2410 <service name="MyService">  
2411   <interface.c header="MyService.h" callbackHeader="MyServiceCallback.h" />  
2412 </service>
```

## 2413 **A.2.5 @OneWay**

2414 Annotation on a service interface function declaration to indicate the function is one way.

2415

2416 **Corresponds to:** oneWay="true" attribute of function element of an interface.c element.

2417

2418 **Format:**

2419 `/* @OneWay */`

2420 The default is **false** (not OneWay).

2421

2422 **Applies to:** Function Prototype



2423

2424 Example

2425 Interface header:

```
2426 /* @OneWay */  
2427 reportEvent(int eventID);
```

2428

2429 Service definition:

```
2430 <service name="LoanService">  
2431   <interface.c header="LoanService.h">  
2432     <operation name="reportEvent" oneWay="true" />  
2433   </interface.c>  
2434 </service>
```

## 2435 A.2.6 @Conversational

2436 Annotation on a service interface to denote a conversational service contract.

2437

2438 **Corresponds to:** requires="conversational" attribute of a service element.

2439

2440 **Format:**

```
2441 /* @Conversational */
```

2442

2443 **Applies to:** Interface

2444

2445 Example

2446 Interface header (LoanService.h):

```
2447 /* @Conversational */
```

2448

2449 Service definition:

```
2450 <service name="LoanService" requires="conversational">  
2451   <interface.c header="LoanService.h">  
2452 </service>
```

## 2453 A.2.7 @EndsConversation

2454 Annotation on a service interface function declaration to indicate that the conversation will be ended when  
2455 this function is called

2456

2457 **Corresponds to:** endsConversation="true" attribute of function element of an interface.c element.

2458

2459 **Format:**

```
2460 /* @EndsConversation */
```

2461 The default is **false** (invoking the function does not end the conversation).

2462

2463 **Applies to:** Function Prototype

2464

2465 Example

2466 Interface header:

```
2467 /* @EndsConversation */
2468 void cancelApplication( );
```

2469

2470 Component definition:

```
2471 <service name="LoanService">
2472   <interface.c header="LoanService.h">
2473     <operation name="cancelApplication" endsConversation="true" />
2474   </interface.c>
2475 </service>
```

## 2476 A.3 Implementation Annotations

2477 This section lists the annotations that may be used in the file that implements a service.

### 2478 A.3.1 @ComponentType

2479 Annotation used to indicate the start of a new componentType.

2480

2481 **Corresponds to:** componentType attribute of implementation.c element.

2482

2483 **Format:**

```
2484 /* @ComponentType */
```

2485

2486 **Applies to:** Set of services, references and properties

2487

2488 Example

2489 Implementation:

```
2490 /* @ComponentType */
```

2491

2492 Component definition:

```
2493 <component name="LoanService">
2494   <implementation.c module="loan"
2495     componentType="LoanService" />
2496 </component>
```

### 2497 A.3.2 @Service

2498 Annotation that indicates the start of a new service implementation.

2499

2500 **Corresponds to:** implementation.c element

2501

2502 **Format:**

```
2503 /* @Service(name="serviceName", interfaceHeader="headerFile") */
```

2504 where

- 2505 • **name : NCName (1..1)** – specifies the name of the service.
- 2506 • **interfaceHeader : Name (1..1)** – specifies the C header defining the interface.

2507

2508 **Applies to:** Set of functions implementing a service

2509 Function definitions following this annotation form the implementation of this service. This annotation also  
2510 serves to bound the scope of the remaining annotations in this section,

2511

2512 Example

2513 Implementation:

```
2514 /* @Service(name="LoanService", interfaceHeader="loan.h") */
```

2515

2516 ComponentType definition:

```
2517 <componentType name="LoanService">  
2518   <service name="LoanService">  
2519     <interface.c header="loans.h" />  
2520   </service>  
2521 </componentType>
```

### 2522 A.3.3 @Reference

2523 Annotation on a service implementation to indicate it depends on another service providing a specified  
2524 interface.

2525

2526 **Corresponds to:** reference element of componentType element.

2527

2528 **Format:**

```
2529 /* @Reference(name="referenceName", interfaceHeader="headerFile",  
2530 *           required="true", multiple="true")  
2531 */
```

2532 where

- 2533 • **name** : *NCName (1..1)* – specifies the name of the reference.
- 2534 • **interfaceHeader** : *Name (1..1)* – specifies the C header defining the interface.
- 2535 • **required** : *boolean (0..1)* – specifies whether a value must has to be set for this reference. Default is  
2536 **true**.
- 2537 • **multiple** : *boolean (0..1)* – specifies whether this reference can be wired to multiple services. Default  
2538 is **false**.

2539

2540 The multiplicity of the reference is determined from the **required** and **multiple** attributes. If the value of  
2541 the **multiple** attribute is true, then component type has a reference with a multiplicity of either 0..n or 1..n  
2542 depending on the value of the **required** attribute – 1..n applies if required=true. Otherwise a multiplicity of  
2543 0..1 or 1..1 is implied.

2544

2545 **Applies to:** Service

2546

2547 Example

2548 Implementation:

```
2549 /* @Reference(name="getRate", interfaceHeader="rates.h") */  
2550  
2551 /* @Reference(name="publishRate", interfaceHeader="myRates.h",  
2552 *           required="false", multiple="yes"  
2553 */
```

2554

2555 ComponentType definition:

```

2556 <componentType name="LoanService">
2557   <reference name="getRate">
2558     <interface.c header="rates.h">
2559   </reference>
2560   <reference name="publishRate" multiplicity="0..n">
2561     <interface.c header="myRates.h">
2562   </reference>
2563 </componentType>

```

### 2564 A.3.4 @Property

2565 Annotation on a service implementation to define a property of the service. Should immediately precedes  
 2566 the global variable that the property is based on. The variable declaration is only used for determining the  
 2567 type of the property. The variable will not be populated with the property value at runtime. Programs use  
 2568 the SCAProperty<Type>() functions for accessing property data.

2569

2570 **Corresponds to:** property element of componentType element.

2571

2572 **Format:**

```

2573 /* @Property (name="propertyName", type="typeName",
2574 *      default="defaultValue", required="true")
2575 */

```

2576 where

- 2577 • **name : NCName (0..1)** – specifies the name of the property. If name is not specified the property  
 2578 name is taken from the name of the global variable.
- 2579 • **type : QName (0..1)** – specifies the type of the property. If not specified the type of the property is  
 2580 based on the C mapping of the type of the following global variable to an xsd type as defined in the  
 2581 appendix Error! Reference source not found.. If the variable is an array, then the property is many-  
 2582 valued.
- 2583 • **required : boolean (0..1)** – specifies whether a value must has to be set in the component definition  
 2584 for this property. Default is **false**.
- 2585 • **default : <type> (0..1)** – specifies a default value and is only needed if **required** is **false**.

2586

2587 **Applies to:** Variable

2588

2589 **Example**

2590 **Implementation:**

```

2591 /* @Property */
2592 long loanType;

```

2593

2594 **ComponentType definition:**

```

2595 <componentType name="LoanService">
2596   <property name="loanType" type="xsd:int" />
2597 </componentType>

```

### 2598 A.3.5 @Scope

2599 Annotation on a service implementation to indicate the scope of the service.

2600

2601 **Corresponds to:** scope attribute of implementation.c element.

2602

2603 **Format:**

```
2604 /* @Scope("value") */
```

2605 where

- 2606 • **value** : [*stateless* | *composite* | *request* | *conversation*] (1..1) – specifies the scope of the  
2607 implementation. The default value is stateless.

2608

2609 **Applies to:** Service

2610

2611 Example

2612 Implementation:

```
2613 /* @Scope("composite") */
```

2614

2615 Component definition:

```
2616 <component name="LoanService">  
2617   <implementation.c module="loan" scope="composite" />  
2618 </component>
```

### 2619 **A.3.6 @Init**

2620 Annotation on a service implementation to indicate a function to be called when the service is  
2621 instantiated. If the service is implemented in a program, this annotation indicates the program is to be  
2622 called with an initialization flag prior to the first operation.

2623

2624 **Corresponds to:** `init="true"` attribute of `implementation.c` element or a function element of an  
2625 `implementation.c` element.

2626

2627 **Format:**

```
2628 /* @Init */
```

2629

2630 **Applies to:** Function or Service

2631

2632 Example

2633 Implementation:

```
2634 /* @Init */  
2635 void init();
```

### 2636 **A.3.7 @Destroy**

2637 Annotation on a service implementation to indicate a function to be called when the service is terminated.  
2638 If the service is implemented in a program, this annotation indicates the program is to be called with an  
2639 termination flag after to the final operation.

2640

2641 **Corresponds to:** `destroy="true"` attribute of `implementation.c` element or a function element of an  
2642 `implementation.c` element.

2643

2644 **Format:**

```
2645 /* @Destroy */
```

2646

2647 **Applies to:** Function or Service

2648

2649 **Example**

2650 Implementation:

```
2651 /* @Destroy */  
2652 void cleanup();
```

### 2653 **A.3.8 @EagerInit**

2654 Annotation on a service implementation to indicate the service is to be instantiated when its containing  
2655 component is started.

2656

2657 **Corresponds to:** eagerInit="true" attribute of implementation.c element.

2658

2659 **Format:**

```
2660 /* @EagerInit */
```

2661

2662 **Applies to:** Service

2663

2664 **Example**

2665 Implementation:

```
2666 /* @EagerInit */
```

2667

2668 Component definition:

```
2669 <component name="LoanService">  
2670   <implementation.c module="loan" eagerInit="true" />  
2671 </component>
```

### 2672 **A.3.9 @AllowsPassByReference**

2673 Annotation on service implementation or operation to indicate that a service or operation allows pass by  
2674 reference semantics.

2675

2676 **Corresponds to:** allowsPassByReference="true" attribute of implementation.c element or a function  
2677 element of an implementation.c element.

2678

2679 **Format:**

```
2680 /* @AllowsPassByReference */
```

2681

2682 **Applies to:** Service or Function

2683

2684 **Example**

2685 Implementation:

```
2686 /* @Service(name="LoanService")  
2687 * @AllowsPassByReference  
2688 */
```

2689

2690 Component definition:

```
2691 <component name="LoanService">
2692   <implementation.c module="loan" allowsPassByReference="true" />
2693 </component>
```

### 2694 **A.3.10 @ConversationAttributes**

2695 Annotation on a service implementation to specify attributes of a conversational service.

2696

2697 **Corresponds to:** conversationMaxAge, conversationMaxIdle or conversationSinglePrincipal attributes of  
2698 implementation.c element.

2699

2700 **Format:**

```
2701 /* @ConversationAttributes(maxIdleTime="value", maxAge="value",
2702    singlePrincipal=boolValue) */
```

2703 where

- 2704 • **maxIdleTime : string (0..1)** – specifies the maximum time that can pass between operations within a  
2705 single conversation. **value** is a time expressed as an integer followed by a space and then one of the  
2706 following: "seconds", "minutes", "hours", "days" or "years".
- 2707 • **maxAge : string (0..1)** – specifies the maximum time that the entire conversation can remain active.  
2708 **value** is a time expressed as an integer followed by a space and then one of the following: "seconds",  
2709 "minutes", "hours", "days" or "years".
- 2710 • **singlePrincipal : boolean (0..1)** – specifies if only the principal (the user) that started the  
2711 conversation has authority to continue the conversation.

2712

2713 **Applies to:** Service

2714

2715 **Example**

2716 **Implementation:**

```
2717 /* @ConversationAttributes(maxAge="30 days", maxIdleTime="5 minutes",
2718    * singlePrincipal=false)
2719    */
```

2720

2721 **Component definition:**

```
2722 <component name="LoanService">
2723   <implementation.c module="loan" conversationMaxAge="30 days"
2724     conversationMaxIdle="5 minutes" conversationSinglePrincipal="false" />
2725 </component>
```

2726

## B Policy Annotations for C

2727 SCA provides facilities for the attachment of policy-related metadata to SCA assemblies, which influence  
2728 how implementations, services and references behave at runtime. The policy facilities are described in  
2729 **[POLICY]**. In particular, the facilities include Intents and Policy Sets, where intents express abstract,  
2730 high-level policy requirements and policy sets express low-level detailed concrete policies.

2731

2732 Policy metadata can be added to SCA assemblies through the means of declarative statements placed  
2733 into Composite documents and into Component Type documents. These annotations are completely  
2734 independent of implementation code, allowing policy to be applied during the assembly and deployment  
2735 phases of application development.

2736

2737 However, it can be useful and more natural to attach policy metadata directly to the code of  
2738 implementations. This is particularly important where the policies concerned are relied on by the code  
2739 itself. An example of this from the Security domain is where the implementation code expects to run  
2740 under a specific security Role and where any service operations invoked on the implementation must be  
2741 authorized to ensure that the client has the correct rights to use the operations concerned. By annotating  
2742 the code with appropriate policy metadata, the developer can rest assured that this metadata is not lost or  
2743 forgotten during the assembly and deployment phases.

2744

2745 The SCA C policy annotations provide the capability for the developer to attach policy information to C  
2746 implementation code. The annotations concerned first provide general facilities for attaching SCA Intents  
2747 and Policy Sets to C code. Secondly, there are further specific annotations that deal with particular policy  
2748 intents for certain policy domains such as Security.

### B.1 General Intent Annotations

2749 SCA provides the annotation **@Requires** for the attachment of any intent to a C function, to a C function  
2750 declaration or to sets of functions implementing a service or sets of function declarations defining a  
2751 service interface.

2752

2753 The **@Requires** annotation can attach one or multiple intents in a single statement.

2754

2755 Each intent is expressed as a string. Intents are XML QNames, which consist of a Namespace URI  
2756 followed by the name of the Intent. The precise form used is as follows:

2757

```
2758 "{ " + Namespace URI + "}" + intentname
```

2759

2760 Intents may be qualified, in which case the string consists of the base intent name, followed by a ".",  
2761 followed by the name of the qualifier. There may also be multiple levels of qualification.

2762

2763 This representation is quite verbose, so we expect that reusable constants will be defined for the  
2764 namespace part of this string, as well as for each intent that is used by C code. SCA defines constants  
2765 for intents such as the following:

2766

```
2767 #define SCA_PREFIX "{http://docs.oasis-open.org/ns/opencsa/sca/200712"  
2768 #define CONFIDENTIALITY SCA_PREFIX ## "confidentiality"  
2769 #define CONFIDENTIALITY_MESSAGE CONFIDENTIALITY ## ".message"
```



2771  
2772 Notice that, by convention, qualified intents include the qualifier as part of the name of the constant,  
2773 separated by an underscore. These intent constants are defined in the file that defines an annotation for  
2774 the intent (annotations for intents, and the formal definition of these constants, are covered in a following  
2775 section).

2776  
2777 Multiple intents (qualified or not) are expressed as separate strings within an array declaration.  
2778

2779 **Corresponds to:** requires attribute of a service, reference, operation or property element.  
2780

2781 **Format:**

```
2782 /* @Requires("qualifiedIntent" | {"qualifiedIntent" [, "qualifiedIntent"]}) */
```

2783 where

```
2784 qualifiedIntent ::= QName | QName.qualifier | QName.qualifier1.qualifier2
```

2785  
2786 **Applies to:** Interface, Service, Function, Function Prototype, Variable  
2787

2788 Examples:

2789 Attaching the intents "confidentiality.message" and "integrity.message".

```
2790 /* @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE}) */
```

2791  
2792 A reference requiring support for confidentiality:

```
2793 /* @Requires(CONFIDENTIALITY)  
2794 * @Reference(interfaceHeader="SetBar.h") */  
2795 void setBar(struct barType *bar);
```

2796  
2797 Users may also choose to only use constants for the namespace part of the QName, so that they may  
2798 add new intents without having to define new constants. In that case, this definition would instead look  
2799 like this:

```
2800  
2801 /* @Requires(SCA_PREFIX "confidentiality")  
2802 * @Reference(interfaceHeader="SetBar.h") */  
2803 void setBar(struct barType *bar);
```

## 2804 B.2 Specific Intent Annotations

2805 In addition to the general intent annotation supplied by the @Requires annotation described above, there  
2806 are C annotations that correspond to some specific policy intents.

2807  
2808 The general form of these specific intent annotations is an annotation with a name derived from the name  
2809 of the intent itself. If the intent is a qualified intent, qualifiers are supplied as an attribute to the annotation  
2810 in the form of a string or an array of strings.

2811  
2812 For example, the SCA confidentiality intent described in [General Intent Annotations](#) using the  
2813 @Requires(CONFIDENTIALITY) intent can also be specified with the specific @Confidentiality intent  
2814 annotation. The specific intent annotation for the "integrity" security intent is:

```
2815  
2816 /* @Integrity */
```

2817

2818 **Corresponds to:** requires="<Intent>" attribute of a service, reference, operation or property element.

2819

2820 **Format:**

```
2821 /* @<Intent>[(qualifiers)] */
```

2822 where Intent is an NCName that denotes a particular type of intent.

```
2823 Intent ::= NCName
```

```
2824 qualifiers ::= "qualifier" | {"qualifier" [, "qualifier" ] }
```

```
2825 qualifier ::= NCName | NCName/qualifier
```

2826

2827 **Applies to:** Interface, Service, Function, Function Prototype, Variable – but see specific intents for  
2828 restrictions

2829

2830 **Example:**

```
2831 /* @Authentication( {"message", "transport"} ) */
```

2832 This annotation attaches the pair of qualified intents: "authentication.message" and

2833 "authentication.transport" (the sca: namespace is assumed in this both of these cases – "http://

2834 docs.oasis-open.org/ns/opencsa/sca/200712").

## 2835 B.3 Application of Intent Annotations

2836 Where multiple intent annotations (general or specific) are applied to the same C element, they are  
2837 additive in effect. An example of multiple policy annotations being used together follows:

2838

```
2839 /* @Authentication
```

```
2840 * @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE}) */
```

2841

2842 In this case, the effective intents are "authentication", "confidentiality.message" and "integrity.message".

2843

2844 If an annotation is specified at both the implementation/interface level and the function or variable level,  
2845 then the function or variable level annotation completely overrides the implementation/interface level  
2846 annotation of the same type.

2847

2848 The intent annotation can be applied either to interface or to functions when adding annotated policy on  
2849 SCA services.

## 2850 B.4 Policy Annotation Scope

2851 The following examples show scope of intents on functions, function declarations and sets of each.

2852

```
2853 /* @Remotable
```

```
2854 * @Integrity("transport")
```

```
2855 * @Authentication
```

```
2856 * @Service(name="HelloService", interfaceHeader="helloworld.h" */
```

```
2857
```

```
2858 /* @Integrity
```

```
2859 * @Authentication("message") */
```

```
2860 wchar_t* hello(wchar_t* message) {...}
```

```
2861
```

```
2862 /* @Integrity
```

```
2863 * @Authentication("transport") */
```

```
2864 wchar_t* helloThere() {...}
```

```

2865
2866 /* @Remotable
2867 * @Confidentiality("message")
2868 * @Service(name="HelloHelperService", interfaceHeader="helloService.h" */
2869
2870 /* @Confidentiality("transport") */
2871 wchar_t* hello(wchar_t* message) {...}
2872
2873 /* @Authentication */
2874 wchar_t* helloWorld(){...}

```

2875 Example 1a. Usage example of annotated policy and inheritance.

2876

- 2877 • The effective intent annotation on the helloWorld function is **@Authentication**, and
- 2878 **@Confidentiality("message")**.
- 2879 • The effective intent annotation on the hello function of the HelloHelperService is
- 2880 **@Confidentiality("transport")**,
- 2881 • The effective intent annotation on the helloThere function of the HelloService is **@Integrity** and
- 2882 **@Authentication("transport")**.
- 2883 • The effective intent annotation on the hello function of the HelloService is **@Integrity** and
- 2884 **@Authentication("message")**

2885

2886 The listing below contains the equivalent declarative security interaction policy of the HelloService and  
 2887 HelloChildService implementation corresponding to the Java interfaces and classes shown in Example  
 2888 1a.

2889

```

2890 <?xml version="1.0" encoding="ASCII"?>
2891
2892 <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
2893           name="HelloServiceComposite" >
2894   <service name="HelloService" requires="integrity/transport
2895           authentication">
2896     ...
2897   </service>
2898   <service name="HelloHelperService" requires="integrity/transport
2899           authentication confidentiality/message">
2900     ...
2901   </service>
2902   ...
2903
2904   <component name="HelloServiceComponent">*
2905     <implementation.c library="HelloService.dll"
2906           header="HelloService.h"/>
2907     <operation name="hello" requires="integrity
2908           authentication/message"/>
2909     <operation name="helloThere" requires="integrity
2910           authentication/transport"/>
2911   </component>
2912   <component name="HelloHelperServiceComponent">*
2913     <implementation.c library="HelloService.dll"
2914           header="HelloService.h" />
2915     <operation name="hello" requires="confidentiality/transport"/>
2916     <operation name="helloWorld" requires="authentication"/>
2917   </component>
2918   ...
2919   ...
2920
2921 </composite>

```

2922 Example 1b. Declaratives intents equivalent to annotated intents in Example 1a.

## 2923 B.5 Relationship of Declarative And Annotated Intents

2924 Annotated intents on a C functions or function declarations cannot be overridden by declarative intents  
2925 either in a composite document which uses the class as an implementation or by statements in a  
2926 component Type document associated with the class. This rule follows the general rule for intents that  
2927 they represent fundamental requirements of an implementation.

2928

2929 An unqualified version of an intent expressed through an annotation in the C function or function  
2930 declaration may be qualified by a declarative intent in a using composite document.

## 2931 B.6 Policy Set Annotations

2932 The SCA Policy Framework uses Policy Sets to capture detailed low-level concrete policies (for example,  
2933 a concrete policy is the specific encryption algorithm to use when encrypting messages when using a  
2934 specific communication protocol to link a reference to a service).

2935

2936 Policy Sets can be applied directly to C implementations using the **@PolicySets** annotation. The  
2937 PolicySets annotation either takes the QName of a single policy set as a string or the name of two or  
2938 more policy sets as an array of strings.

2939

2940 **Corresponds to:** policySets attribute of a service, reference, operation or property element.

2941

2942 **Format:**

```
2943 /* @PolicySets( "<policy set QName>" |  
2944 * { "<policy set QName>" [, "<policy set QName>"] }) */
```

2945 As for intents, PolicySet names are QNames – in the form of "{Namespace-URI}localPart".

2946

2947 **Applies to:** Interface, Service, Function, Function Prototype, Variable

2948

2949 **Example:**

```
2950 /* @Reference(name="helloService", interfaceHeader="helloService.h",  
2951 *           required=true)  
2952 * @PolicySets({ MY_NS "WS_Encryption_Policy",  
2953 *             MY_NS "WS_Authentication_Policy" }) */  
2954 HelloService* helloService;  
2955 ...  
2956 }
```

2957

2958 In this case, the Policy Sets WS\_Encryption\_Policy and WS\_Authentication\_Policy are applied, both  
2959 using the namespace defined for the constant MY\_NS.

2960

2961 PolicySets must satisfy intents expressed for the implementation when both are present, according to the  
2962 rules defined in **[POLICY]**.

## 2963 B.7 Security Policy Annotations

2964 This section introduces annotations for SCA's security intents, as defined in **[POLICY]**.

### 2965 B.7.1 Security Interaction Policy

2966 The following interaction policy Intents and qualifiers are defined for Security Policy, which apply to the  
2967 operation of services and references of an implementation:

- 2968 • @Integrity
- 2969 • @Confidentiality
- 2970 • @Authentication

2971

2972 All three of these intents have the same pair of Qualifiers:

- 2973 • message
- 2974 • transport

2975

2976 The following example shows an example of applying an intent to a reference. Accessing the hello  
 2977 operation of the referenced HelloService requires both "integrity.message" and "authentication.message"  
 2978 intents to be honored.

2979

2980

2981 /\*Interface for HelloService \*/  
 2982 /\* @Interface(name="HelloService") \*/

2983 wchar\_t\* hello(wchar\_t\* helloMsg);

2984

2985 /\* Implementation for ClientService \*/

2986 #include "HelloService.h"

2987

2988 /\* @Service(name="ClientService", interfaceHeader="ClientService.h") \*/

2989

2990 /\* @Integrity("message")

2991 \* @Authentication("message")

2992 \* @Reference(name="helloService", interfaceHeader="HelloService.h") \*/

2993

2994 void clientMethod() {

2995 SCAREF serviceToken;

2996 int compCode, reason;

2997 wchar\_t result[256];

2998

2999 SCALocate(L"customerService", &serviceToken, &compCode, &reason);

3000 SCAInvoke(serviceToken, L"hello", sizeof(L"Hello World!"),

3001 L"Hello World!", sizeof(result), (void \*)&result,

3002 &compCode, &reason);

3003 }

## 3003 B.7.2 Security Implementation Policy

3004 SCA defines a number of security policy annotations that apply as policies to implementations  
 3005 themselves. These annotations mostly have to do with authorization and security identity. The following  
 3006 authorization and security identity annotations are supported:

- 3007 • @RunAs

3008 Takes as a parameter a string which is the name of a Security role, e.g.

3009 @RunAs("Manager").

3010 Code marked with this annotation will execute with the Security permissions of the  
 3011 identified role.

- 3012 • @RolesAllowed

3013 Takes as a parameter a single string or an array of strings which represent one or more  
 3014 role names. When present, the implementation can only be accessed by principals  
 3015 whose role corresponds to one of the role names listed in the @DeclareRoles attribute),

3016 e.g. @RolesAllowed( {"Manager", "Employee"} ).

3017 How role names are mapped to security principals is implementation dependent (SCA  
3018 does not define this

3019 • @PermitAll

3020 No parameters. When present, grants access to all roles.

3021 • @DenyAll

3022 No parameters. When present, denies access to all roles.

3023 • @DeclareRoles

3024 Takes as a parameter a string or an array of strings which identify one or more role  
3025 names that form the set of roles used by the implementation,

3026 e.g. @DeclareRoles({"Manager", "Employee", "Customer"})

3027

3028 For a full explanation of these intents, see [POLICY].

### 3029 **B.7.2.1 Annotated Implementation Policy Example**

3030 The following is an example showing annotated security implementation policy:

3031

```
3032 /* @Remotable  
3033 * @Interface(name="AccountService") */  
3034 struct AccountReport *getAccountReport(char *customerID);
```

3035

3036 The following is a full listing of the AccountService implementation, showing the Service it implements,  
3037 plus the service references it makes and the settable properties that it has, along with a set of  
3038 implementation policy annotations:

3039

```
3040 #include "SCA.h"  
3041 #include "AccountService.h";  
3042 #include "AccountDataService.h";  
3043 #include "StockQuoteService.h";  
3044  
3045 /* @RolesAllowed("customers")  
3046 * @RunAs("accountants" )  
3047 * @Service(name="AccountService", interfaceHeader="AccountService.h") */  
3048  
3049 /* @Property */  
3050 char currency[5] = "USD";  
3051  
3052 /* @Reference(name="accountDataService",  
3053 interfaceHeader="AccountDataService.h") */  
3054  
3055 /* @Reference(name="stockQuoteService",  
3056 interfaceHeader="StockQuoteService.h") */  
3057  
3058 /* @RolesAllowed({"customers", "accountants"}) */  
3059 struct AccountReport *getAccountReport(char *customerID) {  
3060 struct AccountReport *accounts;  
3061 struct CheckingAccount *checking;  
3062 struct SavingsAccount *savings;  
3063 struct StockAccount *stock;  
3064 float balance;  
3065 struct QuoteRequest stockToQuote;  
3066 SCAREF, accountDataService, stockQuoteService;  
3067 int compCode, Reason;  
3068
```

```

3069     accounts = (struct AccountReport *)malloc(sizeof(struct AccountReport));
3070     accounts->numAccounts = 0;
3071
3072     SCALocate(L"accountDataService", &accountDataService, &compCode, &Reason);
3073
3074     SCAInvoke(accountDataService, L"getCheckingAccount",
3075             sizeof(customerID), (void *)customerID,
3076             sizeof(struct CheckingAccount), (void *)checking,
3077             &compCode, &Reason);
3078     accounts->numAccounts += 1;
3079     strcpy(accounts->summary[0].number, checking->number);
3080     strcpy(accounts->summary[0].type, "checking");
3081     accounts->summary[0].balance = fromUSDollarToCurrency(checking->balance);
3082
3083     SCAInvoke(accountDataService, L"getSavingsAccount",
3084             sizeof(customerID), (void *)customerID,
3085             sizeof(struct SavingsAccount), (void *)savings,
3086             &compCode, &Reason);
3087     accounts->numAccounts += 1;
3088     strcpy(accounts->summary[1].number, savings->number);
3089     strcpy(accounts->summary[1].type, "savings");
3090     accounts->summary[1].balance = fromUSDollarToCurrency(savings->balance);
3091
3092     SCAInvoke(accountDataService, L"getStockAccount",
3093             sizeof(customerID), (void *)customerID,
3094             sizeof(struct StockAccount), (void *)stock,
3095             &compCode, &Reason);
3096     accounts->numAccounts += 1;
3097     strcpy(accounts->summary[2].number, stock->number);
3098     strcpy(accounts->summary[2].type, "stock");
3099     strcpy(stockToQuote.symbol, stock->symbol);
3100     stockToQuote.quantity = stock->quantity;
3101     SCALocate(L"stockQuoteService", &stockQuoteService, &compCode, &Reason);
3102     SCAInvoke(stockQuoteService, L"getQuote",
3103             sizeof(struct QuoteRequest), (void *)&stocktoQuote,
3104             sizeof(balance), (void *)&balance,
3105             &compCode, &Reason);
3106     accounts->summary[2].balance = fromUSDollarToCurrency(balance);
3107
3108     return accounts;
3109 }
3110
3111 /* @PermitAll */
3112 float fromUSDollarToCurrency(float value){
3113
3114     if (!strcmp(currency, "USD"))
3115         return value;
3116     else if (!strcmp(currency, "EURO"))
3117         return value * 0.8;
3118     else
3119         return 0.0;
3120 }

```

3121  
3122 In this example, the implementation as a whole is marked:

- 3123 • @RolesAllowed("customers") - indicating that customers have access to the implementation as a
- 3124 whole
- 3125 • @RunAs("accountants") – indicating that the code in the implementation runs with the permissions of
- 3126 accountants

3127  
3128 The getAccountReport(..) funcgion is marked with @RolesAllowed({"customers", "accountants"}), which

3129 indicates that this function can be called by both customers and accountants.

3130

3131 The fromUSDollarToCurrency() function is marked with @PermitAll, which means that this function can  
3132 be called by any role.



---

## 3133 C XML Schemas

3134 Two XML schemas are defined to support the use of C for implementation and definition of interfaces.

3135 The normative schemas are located at:

3136 <http://docs.oasis-open.org/opencsa/sca-c-cpp/sca-interface-c-1.1-schema.xsd>

3137 and

3138 <http://docs.oasis-open.org/opencsa/sca-c-cpp/sca-implementation-c-1.1-schema.xsd>

3139

3140 The following copies are provided for reference.

### 3141 C.1 sca-interface-c-1.1-schema.xsd

```
3142 <?xml version="1.0" encoding="UTF-8"?>
3143 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3144         targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3145         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3146         elementFormDefault="qualified">
3147
3148     <include schemaLocation="sca-core.xsd"/>
3149
3150     <element name="interface.c" type="sca:CInterface"
3151             substitutionGroup="sca:interface"/>
3152
3153     <complexType name="CInterface">
3154         <complexContent>
3155             <extension base="sca:Interface">
3156                 <sequence>
3157                     <element name="function" type="sca:CFunction"
3158                             minOccurs="0" maxOccurs="unbounded" />
3159                     <element name="callbackFunction" type="sca:CFunction"
3160                             minOccurs="0" maxOccurs="unbounded" />
3161                     <any namespace="##other" processContents="lax"
3162                             minOccurs="0" maxOccurs="unbounded"/>
3163                 </sequence>
3164                 <attribute name="header" type="Name" use="required"/>
3165                 <attribute name="callbackHeader" type="Name" use="optional"/>
3166                 <attribute name="remotable" type="boolean" use="optional"/>
3167                 <anyAttribute namespace="##other" processContents="lax"/>
3168             </extension>
3169         </complexContent>
3170     </complexType>
3171
3172     <complexType name="CFunction">
3173         <attribute name="name" type="NCName" use="required"/>
3174         <attribute name="oneWay" type="boolean" use="optional"/>
3175         <attribute name="endsConversation" type="boolean" use="optional"/>
3176         <attribute name="input" type="NCName" use="optional"/>
3177         <attribute name="output" type="NCName" use="optional"/>
3178         <anyAttribute namespace="##other" processContents="lax"/>
3179     </complexType>
3180
3181 </schema>
```

### 3182 C.2 sca-implementation-c-1.1-schema.xsd

```
3183 <?xml version="1.0" encoding="UTF-8"?>
3184 <schema xmlns="http://www.w3.org/2001/XMLSchema"
```

```

3185     targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3186     xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
3187     elementFormDefault="qualified">
3188
3189     <include schemaLocation="sca-core.xsd"/>
3190
3191     <element name="implementation.c" type="sca:CImplementation"
3192         substitutionGroup="sca:implementation" />
3193
3194     <complexType name="CImplementation">
3195         <complexContent>
3196             <extension base="sca:Implementation">
3197                 <sequence>
3198                     <element name="operation" type="sca:CImplementationFunction"
3199                         minOccurs="0" maxOccurs="unbounded" />
3200                     <any namespace="##other" processContents="lax"
3201                         minOccurs="0" maxOccurs="unbounded"/>
3202                 </sequence>
3203                 <attribute name="module" type="NCName" use="required"/>
3204                 <attribute name="location" type="Name" use="optional"/>
3205                 <attribute name="library" type="boolean" use="optional"/>
3206                 <attribute name="componentType" type="Name"/>
3207                 <attribute name="scope" type="sca:CImplementationScope"
3208                     use="optional"/>
3209                 <attribute name="eagerInit" type="boolean" use="optional"/>
3210                 <attribute name="init" type="boolean" use="optional"/>
3211                 <attribute name="destroy" type="boolean" use="optional"/>
3212                 <attribute name="allowsPassByReference" type="boolean"
3213                     use="optional"/>
3214                 <attribute name="conversationMaxAge" type="string"
3215                     use="optional"/>
3216                 <attribute name="conversationMaxIdle" type="string"
3217                     use="optional"/>
3218                 <attribute name="conversationSinglePrincipal" type="boolean"
3219                     use="optional"/>
3220                 <anyAttribute namespace="##other" processContents="lax"/>
3221             </extension>
3222         </complexContent>
3223     </complexType>
3224
3225     <simpleType name="CImplementationScope">
3226         <restriction base="string">
3227             <enumeration value="stateless"/>
3228             <enumeration value="composite"/>
3229             <enumeration value="request"/>
3230             <enumeration value="converstion"/>
3231         </restriction>
3232     </simpleType>
3233
3234     <complexType name="CImplementationFunction">
3235         <attribute name="name" type="NCName" use="required"/>
3236         <attribute name="allowsPassByReference" type="boolean"
3237             use="optional"/>
3238         <attribute name="init" type="boolean" use="optional"/>
3239         <attribute name="destroy" type="boolean" use="optional"/>
3240         <anyAttribute namespace="##other" processContents="lax"/>
3241     </complexType>
3242
3243 </schema>

```

---

3244 **D Migration**

3245 To aid migration of an implementation or clients using an implementation based the version of the Service  
3246 Component Architecture for C defined in [SCA C Client and Implementation V1.00](#), this appendix identifies  
3247 the relevant changes to APIs, annotations, or behavior defined in V1.00.

3248 **D.1 Annotations related to conversations**

3249 @Conversation has been changed to @ConversationAttributes. @Conversation is removed.

3250 @EndConversation has been changed to @EndsConversation. @EndConversation is removed.

3251

---

## E Acknowledgements

3252 The following individuals have participated in the creation of this specification and are gratefully  
3253 acknowledged:

3254 **Participants:**

3255 Andrew Borley, IBM

3256 Bryan Aupperle, IBM

3257 David Haney, Rogue Wave Software

3258 Mike Edwards, IBM

3259 Pete Robbins, IBM

3260

---

## **F Non-Normative Text**

3262

## G Revision History

3263 [optional; should not be included in OASIS Standards]

3264

Revision	Date	Editor	Changes Made
5	2008-3-17	Bryan Aupperle	<ul style="list-style-type: none"><li>• Editorial changes in preparation for Committee Draft</li></ul>
4	2008-2-26	Bryan Aupperle	<ul style="list-style-type: none"><li>• Remove duplicated text from Assembly Spec</li><li>• Reformat pseudo-schema presentation and description to be consistent with assembly spec</li><li>• Incorporate changes for CCPP-32, CCPP-34 and CCPP-35</li></ul>
3	2008-1-24	Bryan Aupperle	<ul style="list-style-type: none"><li>• Conformance statement cleanup</li><li>• Incorporate changes for CCPP-20, CCPP-27 and equivalent work for CCPP-31</li></ul>
2	2007-11-08	Bryan Aupperle	<ul style="list-style-type: none"><li>• Update MUST/must, SHOULD/should and MAY/may to reflect TC use or RFC 2119</li><li>• Incorporate changes for CCPP-7, CCPP-8, CCPP-11, CCPP-16 and CCPP-18</li></ul>
1	2007-09-28	Bryan Aupperle	<ul style="list-style-type: none"><li>• Initial conversion of the OSOA 1.0 specification to the OASIS template.</li></ul>

3265