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

Committee Draft 02

5 February 2009

Specification URIs:
This Version:
http://docs.oasis-open.org/opencsa/sca-c-cpp/sca-cppcni-1.1-spec-cd02.html
http://docs.oasis-open.org/opencsa/sca-c-cpp/sca-cppcni-1.1-spec-cd02.doc

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

Latest Version:
http://docs.oasis-open.org/opencsa/sca-c-cpp/sca-cppcni-1.1-spec.html
http://docs.oasis-open.org/opencsa/sca-c-cpp/sca-cppcni-1.1-spec.doc

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

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

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

Related work:
This specification replaces or supercedes:
• OSOA SCA C++ Client and Implementation V1.00

This specification is related to:
• OASIS Service Component Architecture Assembly Model Version 1.1
• OASIS SCA Policy Framework Version 1.1
• OASIS Service Component Architecture Web Service Binding Specification Version 1.1

Declared XML Namespace(s):
http://docs.oasis-open.org/ns/opencsa/sca/200712
http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901

Abstract:
This document describes the SCA Client and Implementation Model for the C++ programming language.
The SCA C++ implementation model describes how to implement SCA components in C++. A component implementation itself can also be a client to other services provided by other components or external services. The document describes how a C++ implemented component gets access to services and calls their operations.

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

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

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

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

1 Introduction .................................................................................................................................8
  1.1 Terminology ..........................................................................................................................8
  1.2 Normative References ...........................................................................................................8
  1.3 Non-Normative References ....................................................................................................9
  1.4 Conventions ..........................................................................................................................9
    1.4.1 Naming Conventions .......................................................................................................9
    1.4.2 Typographic Conventions .............................................................................................9
2 Basic Component Implementation Model ..................................................................................10
  2.1 Implementing a Service ........................................................................................................10
    2.1.1 Implementing a Remotable Service ............................................................................11
    2.1.2 Implementing a Local Service ....................................................................................12
  2.2 Component Implementation Scopes ....................................................................................12
    2.2.1 Stateless scope ...........................................................................................................12
    2.2.2 Composite scope ........................................................................................................13
  2.3 Implementing a Configuration Property ...............................................................................13
  2.4 Component Type and Component .....................................................................................13
    2.4.1 Interface.cpp .............................................................................................................15
    2.4.2 Function and CallbackFunction ................................................................................15
    2.4.3 Implementation.cpp ..................................................................................................16
    2.4.4 Implementation Function ..........................................................................................17
  2.5 Instantiation ........................................................................................................................17
3 Basic Client Model ....................................................................................................................18
  3.1 Accessing Services from Component Implementations ......................................................18
  3.2 Accessing Services from non-SCA component implementations .......................................19
  3.3 Calling Service Operations ................................................................................................19
4 Asynchronous Programming ....................................................................................................20
  4.1 Non-blocking Calls ..............................................................................................................20
  4.2 Callbacks ............................................................................................................................20
    4.2.1 Using Callbacks ...........................................................................................................21
    4.2.2 Callback Instance Management ................................................................................22
    4.2.3 Implementing Multiple Bidirectional Interfaces ........................................................22
5 Error Handling ..........................................................................................................................23
6 C++ API .....................................................................................................................................24
  6.1 Reference Counting Pointers ..............................................................................................24
    6.1.1 operator* ....................................................................................................................24
    6.1.2 operator-> .......................................................................................................24
    6.1.3 operator void* ............................................................................................................25
    6.1.4 operator! ....................................................................................................................25
  6.2 Component Context ..............................................................................................................25
    6.2.1 getCurrent ..................................................................................................................26
    6.2.2 getURI .......................................................................................................................26
    6.2.3 getService ...................................................................................................................26
    6.2.4 getServices ................................................................................................................26
11.1 Conformance Targets ......................................................................................................................43
11.2 Conformance Claims ..........................................................................................................................43
11.3 SCA Implementations ..........................................................................................................................43
11.4 SCDL Documents ................................................................................................................................43
11.5 C++ Component Implementations .........................................................................................................43
11.6 C++ Header Files ..................................................................................................................................43
11.7 WSDL Files ............................................................................................................................................44
11.8 Extensions ............................................................................................................................................44
A C++ SCA Annotations .................................................................................................................................45
A.1 Application of Annotations to C++ Program Elements ........................................................................45
A.2 Interface Header Annotations ................................................................................................................45
  A.2.1 @Interface .........................................................................................................................................46
  A.2.2 @Removable ......................................................................................................................................46
  A.2.3 @Callback ........................................................................................................................................47
  A.2.4 @OneWay .......................................................................................................................................47
A.3 Implementation Header Annotations ........................................................................................................48
  A.3.1 @ComponentType ..........................................................................................................................48
  A.3.2 @Scope ..........................................................................................................................................48
  A.3.3 @EagerInit ......................................................................................................................................49
  A.3.4 @AllowsPassByReference ..............................................................................................................49
  A.3.5 @Property ......................................................................................................................................50
  A.3.6 @Reference ..................................................................................................................................51
A.4 Base Annotation Grammar .........................................................................................................................51
B C++ SCA Policy Annotations .......................................................................................................................53
B.1 General Intent Annotations .......................................................................................................................53
B.2 Specific Intent Annotations .......................................................................................................................54
  B.2.1 Security Interaction ......................................................................................................................55
  B.2.2 Security Implementation ..............................................................................................................55
  B.2.3 Reliable Messaging .......................................................................................................................56
  B.2.4 Transactions ..................................................................................................................................56
  B.2.5 Miscellaneous ................................................................................................................................56
B.3 Application of Intent Annotations ...........................................................................................................56
B.4 Inheritance and Intent Annotations ..........................................................................................................57
B.5 Relationship of Declarative and Annotated Intents ...................................................................................58
B.6 Policy Set Annotations ............................................................................................................................58
B.7 Policy Annotation Grammar Additions ...................................................................................................59
B.8 Annotation Constants ...............................................................................................................................59
C C++ WSDL Mapping Annotations ..............................................................................................................60
C.1 Interface Header Annotations ................................................................................................................60
  C.1.1 @WebService .................................................................................................................................60
  C.1.2 @WebMethod .................................................................................................................................61
  C.1.3 @OneWay ......................................................................................................................................63
  C.1.4 @WebParam ..................................................................................................................................64
  C.1.5 @WebResult ..................................................................................................................................66
  C.1.6 @SOAPBinding ..............................................................................................................................68
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.1.7</td>
<td>@WebFault</td>
<td>69</td>
</tr>
<tr>
<td>C.1.8</td>
<td>@WebThrows</td>
<td>71</td>
</tr>
<tr>
<td>D</td>
<td>WSDL C++ Mapping Extensions</td>
<td>72</td>
</tr>
<tr>
<td>D.1</td>
<td><code>&lt;cpp:bindings&gt;</code></td>
<td>72</td>
</tr>
<tr>
<td>D.2</td>
<td><code>&lt;cpp:class&gt;</code></td>
<td>72</td>
</tr>
<tr>
<td>D.3</td>
<td><code>&lt;cpp:enableWrapperStyle&gt;</code></td>
<td>73</td>
</tr>
<tr>
<td>D.4</td>
<td><code>&lt;cpp:namespace&gt;</code></td>
<td>74</td>
</tr>
<tr>
<td>D.5</td>
<td><code>&lt;cpp:memberFunction&gt;</code></td>
<td>75</td>
</tr>
<tr>
<td>D.6</td>
<td><code>&lt;cpp:parameter&gt;</code></td>
<td>76</td>
</tr>
<tr>
<td>D.7</td>
<td>JAX-WS WSDL Extensions</td>
<td>78</td>
</tr>
<tr>
<td>D.8</td>
<td>WSDL Extensions Schema</td>
<td>79</td>
</tr>
<tr>
<td>E</td>
<td>XML Schemas</td>
<td>81</td>
</tr>
<tr>
<td>E.1</td>
<td><code>sca-interface-cpp-1.1.xsd</code></td>
<td>81</td>
</tr>
<tr>
<td>E.2</td>
<td><code>sca-implementation-cpp-1.1.xsd</code></td>
<td>82</td>
</tr>
<tr>
<td>E.3</td>
<td><code>sca-contribution-cpp-1.1.xsd</code></td>
<td>82</td>
</tr>
<tr>
<td>F</td>
<td>Conformance Items</td>
<td>84</td>
</tr>
<tr>
<td>F.1</td>
<td>JAX-WS Conformance</td>
<td>87</td>
</tr>
<tr>
<td>F.1.1</td>
<td>Ignored Conformance Statements</td>
<td>89</td>
</tr>
<tr>
<td>G</td>
<td>Migration</td>
<td>91</td>
</tr>
<tr>
<td>G.1</td>
<td>Method child elements of interface.cpp and implementation.cpp</td>
<td>91</td>
</tr>
<tr>
<td>H</td>
<td>Acknowledgements</td>
<td>92</td>
</tr>
<tr>
<td>I</td>
<td>Non-Normative Text</td>
<td>93</td>
</tr>
<tr>
<td>J</td>
<td>Revision History</td>
<td>94</td>
</tr>
</tbody>
</table>
1 Introduction

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

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

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

1.1 Terminology

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

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

Table 1-1 Prefixes and Namespaces used in this specification

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Namespace</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>xs</td>
<td>&quot;<a href="http://www.w3.org/2001/XMLSchema">http://www.w3.org/2001/XMLSchema</a>&quot;</td>
<td>Defined by XML Schema 1.0 specification</td>
</tr>
<tr>
<td>sca</td>
<td>&quot;<a href="http://docs.oasis-open.org/ns/opencsa/sca/200712">http://docs.oasis-open.org/ns/opencsa/sca/200712</a>&quot;</td>
<td>Defined by the SCA specifications</td>
</tr>
<tr>
<td>cpp</td>
<td>&quot;<a href="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901">http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901</a>&quot;</td>
<td></td>
</tr>
</tbody>
</table>

1.2 Normative References


1.3 Non-Normative References

1.4 Conventions

1.4.1 Naming Conventions

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

• For the names of elements and the names of attributes within XSD files, the names follow the CamelCase convention, with all names starting with a lower case letter.
  e.g. `<element name="componentType" type="sca:ComponentType"/>

• For the names of types within XSD files, the names follow the CamelCase convention with all names starting with an upper case letter
  e.g. `<complexType name="ComponentService">

• For the names of intents, the names follow the CamelCase convention, with all names starting with a lower case letter, EXCEPT for cases where the intent represents an established acronym, in which case the entire name is in upper case.
  An example of an intent which is an acronym is the "SOAP" intent.

1.4.2 Typographic Conventions

This specification follows some typographic conventions for some specific constructs

• XML attributes are identified in text as @attribute

• Language identifiers used in text are in courier

• Literals in text are in italics
2 Basic Component Implementation Model

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

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

2.1 Implementing a Service

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

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

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

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

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

Service interface.

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

Implementation declaration header file.

```cpp
class LoanServiceImpl : public LoanService {
  public:
    LoanServiceImpl();
    virtual ~LoanServiceImpl();
    virtual bool approveLoan(unsigned long customerNumber,
                             unsigned long loanAmount);
};
```

Implementation.

```cpp
#include "LoanServiceImpl.h"
```
LoanServiceImpl::LoanServiceImpl()
{
    ...
}

LoanServiceImpl::~LoanServiceImpl()
{
    ...
}

bool LoanServiceImpl::approveLoan(unsigned long customerNumber,
    unsigned long loanAmount)
{
    ...
}

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

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

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

2.1.1 Implementing a Remotable Service

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

```xml
<?xml version="1.0" encoding="ASCII"?>
```
Complex data types exchanged via remotable service interfaces MUST be compatible with the marshalling technology that is used by the service binding.

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

2.1.2 Implementing a Local Service

A service interface not marked as remotable is local.

2.2 Component Implementation Scopes

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

Scopes are specified using the @scope attribute of the implementation.cpp element. When a scope is not specified on an implementation class, the SCA runtime will interpret the implementation scope as stateless.

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

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

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

2.2.1 Stateless scope

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

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

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

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

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

2.3 Implementing a Configuration Property

Component implementations can be configured through properties. The properties and their types (not their values) are defined in the component type file. The C++ component can retrieve the properties using the \texttt{getProperties()} on the \texttt{ComponentContext} class.

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

```cpp
#include "ComponentContext.h"
using namespace oasis::sca;

void clientFunction()
{
  ...
  ComponentContext context = ComponentContext::getCurrent();
  DataObjectPtr properties = context.getProperties();
  long loanRating = properties->getInteger("maxLoanValue");
  ...
}
```

2.4 Component Type and Component

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

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

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

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

Implementation declaration header file.

class LoanServiceImpl : public LoanService {
public:
    LoanServiceImpl();
    virtual ~LoanServiceImpl();
    virtual bool approveLoan(unsigned long customerNumber,
        unsigned long loanAmount);
};

Implementation.

#include "LoanServiceImpl.h"

/////////////////////////////////////////////////////////////////////
// Construction/Destruction
/////////////////////////////////////////////////////////////////////
LoanServiceImpl::LoanServiceImpl()
{
...
}
LoanServiceImpl::~LoanServiceImpl()
{
...
}

/////////////////////////////////////////////////////////////////////
// Implementation
/////////////////////////////////////////////////////////////////////
bool LoanServiceImpl::approveLoan(unsigned long customerNumber,
    unsigned long loanAmount)
{
...
}

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

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

The following snippet shows the component using the implementation.

<?xml version="1.0" encoding="ASCII"?>
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
    name="LoanComposite" >
    ...
    <component name="LoanService">
        <implementation.cpp library="loan" class="LoanServiceImpl"/>
    </component>
</composite>
2.4.1 Interface.cpp

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

```xml
<?xml version="1.0" encoding="ASCII"?>
<!-- interface.cpp schema snippet -->
<interface.cpp xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
              header="string" class="Name"? remotable="boolean"?
              callbackHeader="string" callbackClass="Name"? >
  <function ... />*
  <callbackFunction ... />*
</interface.cpp>
```

The `interface.cpp` element has the following attributes:

- **header**: `string (1..1)` – full name of the header file that describes the interface, including relative path from the composite root.
- **class**: `Name (0..1)` – name of the class declaration for the interface in the header file, including any namespace definition. If the header file identified by the `@header` attribute of an `<interface.cpp/>` element contains more than one class, then the `@class` attribute MUST be specified for the `<interface.cpp/>` element. [CPP20005]
- **callbackHeader**: `string (0..1)` – full name of the header file that describes the callback interface, including relative path from the composite root.
- **callbackClass**: `Name (0..1)` – name of the class declaration for the callback interface in the callback header file, including any namespace definition. If the header file identified by the `@callbackHeader` attribute of an `<interface.cpp/>` element contains more than one class, then the `@callbackClass` attribute MUST be specified for the `<interface.cpp/>` element. [CPP20006]
- **remotable**: `boolean (0..1)` – indicates whether the service is remotable or local. The default is local.

See Implementing a Remotable Service

The `interface.cpp` element has the following child elements:

- **function**: `CPPFunction (0..n)` – see Function and CallbackFunction
- **callbackFunction**: `CPPFunction (0..n)` – see Function and CallbackFunction

2.4.2 Function and CallbackFunction

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

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

```xml
<?xml version="1.0" encoding="ASCII"?>
<!-- Function schema snippet -->
<interface.cpp xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" ... >
  <function name="NCName" oneWay="Boolean"? />*
  <callbackFunction name="NCName" oneWay="Boolean"? />*
</interface.cpp>
```
The `function` and `callbackFunction` elements have the following attributes:

- **name**: NCName (1..1) – name of the method being decorated. The `@name` attribute of a `<function/>` child element of a `<interface.cpp/>` MUST be unique amongst the `<function/>` elements of that `<interface.cpp/>`. [CPP20007]

- **oneWay**: boolean (0..1) – see Non-blocking Calls

### 2.4.3 Implementation.cpp

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

```xml
<implementation.cpp xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
library="NCName" path="string"? class="Name"
scope="scope"? componentType="string"? allowsPassByReference="Boolean"?
eagerInit="boolean"?/>
```

The `implementation.cpp` element has the following attributes:

- **library**: NCName (1..1) – name of the dll or shared library that holds the factory for the service component. This is the root name of the library.

- **path**: string (0..1) - path to the library which is either relative to the root of the contribution containing the composite or is prefixed with a contribution import name and is relative to the root of the import. See C++ Contributions.

- **class**: Name (1..1) – name of the class declaration of the implementation, including any namespace definition. The name of the componentType file for a C++ implementation MUST match the class name (excluding any namespace definition) of the implementations as defined by the `@class` attribute of the `<implementation.cpp/>` element. [CPP20009] The SCA runtime will append `.componentType` to the class name to find the componentType file.

- **scope**: CPPImplementationScope (0..1) – identifies the scope of the component implementation. The default is stateless. See Component Implementation Scopes

- **componentType**: string (0..1) – path to the componentType file which is relative to the root of the contribution containing the composite or is prefixed with a contribution import name and is relative to the root of the import.

- **allowsPassByReference**: boolean (0..1) – indicates the service allows pass by reference data exchange semantics on calls to it. See Implementing a Remotable Service

- **eagerInit type**: boolean (0..1) – indicates a composite scoped implementation should be initialized when it is loaded. See Composite scope

The `implementation.cpp` element has the following child element:

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

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

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

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

The `function` element has the following attributes:

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

2.5 Instantiation

A C++ implementation class MUST be default constructable by the SCA runtime to instantiate the component. [CPP20011]
3 Basic Client Model

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

3.1 Accessing Services from Component Implementations

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

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

```cpp
namespace oasis {
    namespace sca {

        class ComponentContext {
        public:
            static ComponentContextPtr getCurrent();
            virtual void * getService(const std::string& referenceName) const = 0;
            ...}
    }
}
```

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

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

```cpp
#include "ComponentContext.h"
#include "CustomerService.h"

using namespace oasis::sca;

void clientFunction() {
    unsigned long customerNumber = 1234;
    ComponentContextPtr context = ComponentContext::getCurrent();
    CustomerService* service = (CustomerService* )context->getService("customerService");
    short rating = service->getCreditRating(customerNumber);
}
```
3.2 Accessing Services from non-SCA component implementations

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

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

3.3 Calling Service Operations

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

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

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

4.1 Non-blocking Calls

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

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

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

```xml
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
  <service name="LoanService">
    <interface.cpp header="LoanService.h">
      <function name="reportEvent" oneWay="true" />
    </interface.cpp>
  </service>
</componentType>
```

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

4.2 Callbacks

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

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

```xml
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
  <service name="MyService">
    <interface.cpp header="MyService.h"
      callbackHeader="MyServiceCallback.h"/>
  </service>
</componentType>
```
4.2.1 Using Callbacks

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

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

```cpp
class Quotation {
public:
    virtual double requestQuotation(std::string productCode,
                                      unsigned int quantity) = 0;
};

class QuotationCallback {
public:
    virtual std::string getState() = 0;
    virtual std::string getZipCode() = 0;
    virtual std::string getCreditRating() = 0;
};
```

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

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

```cpp
#include "QuotationImpl.h"
#include "QuotationCallback.h"
#include "oasis/sca/ComponentContext.h"
using namespace oasis::sca;

double QuotationImpl::requestQuotation(std::string productCode,
                                         unsigned int quantity) {
    double price = getPrice(productQuote, quantity);
    double discount = 0;

    ComponentContextPtr context = ComponentContext::getCurrent();
    ServiceReferencePtr serviceRef = context->getSelfReference();
    QuotationCallback* callback =
        (QuotationCallback* ) serviceRef->getCallback();
    if (quantity > 1000 && callback->getState().compare("FL") == 0)
        discount = 0.05;
    if (quantity > 10000 && callback->getCreditRating().data() == 'A')
        discount += 0.05;
    return price * (1-discount);
}
```
The code snippet below is taken from the client of this example service. The client's service implementation class implements the methods of the QuotationCallback interface as well as those of its own service interface ClientService.

```c++
#include "QuotationImpl.h"
#include "QuotationCallback.h"
#include "oasis/sca/ComponentContext.h"
using namespace oasis::sca;

void ClientImpl:: aClientFunction() {
    ComponentContextPtr context = ComponentContext::getCurrent();
    service = (QuotationService* )context->getService("quotationService");
    service->requestQuotation("AB123", 2000);
}

std::string QuotationCallbackImpl::getState() {
    return "TX";
}

std::string QuotationCallbackImpl::getZipCode() {
    return "78746";
}

std::string QuotationCallbackImpl::getCreditRating() {
    return "AA";
}
```

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

### 4.2.2 Callback Instance Management

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

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

### 4.2.3 Implementing Multiple Bidirectional Interfaces

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

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

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

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

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

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

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

6.1 Reference Counting Pointers

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

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

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

```cpp
template <typename T>
class RefCountingPointer {
public:
    T& operator* () const;
    T* operator-> () const;
    operator void* () const;
    bool operator! () const;
};
```

The `RefCountingPointer` class has the following member functions:

6.1.1 operator`

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

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

6.1.2 operator->

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

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

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

### 6.1.4 operator!

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

<table>
<thead>
<tr>
<th>Precondition</th>
<th>C++ component instance is running and has a reference counting pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return</td>
<td>Zero if the underlying pointer is null, otherwise a non-zero value</td>
</tr>
<tr>
<td>Post Condition</td>
<td>No change</td>
</tr>
</tbody>
</table>

### 6.2 Component Context

The following shows the `ComponentContext` interface.

```cpp
class ComponentContext {
public:
    static ComponentContextPtr getCurrent();
    virtual std::string getURI() const = 0;
    virtual void* getService(const std::string& referenceName) const = 0;
    virtual std::list<void*> getServices(const std::string& referenceName) const = 0;
    virtual ServiceReferencePtr getServiceReference(const std::string& referenceName) const = 0;
    virtual std::list<ServiceReferencePtr> getServiceReferences(const std::string& referenceName) const = 0;
    virtual DataObjectPtr getProperties() const = 0;
    virtual DataFactoryPtr getDataFactory() const = 0;
    virtual ServiceReferencePtr getSelfReference() const = 0;
    virtual ServiceReferencePtr getSelfReference(const std::string& serviceName) const = 0;
};
```
The `ComponentContext` C++ interface has the following member functions:

### 6.2.1 `getCurrent`

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

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

### 6.2.2 `getURI`

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

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

### 6.2.3 `getService`

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

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

### 6.2.4 `getServices`

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

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

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

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

### 6.2.6 `getServiceReferences`

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

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

### 6.2.7 `getProperties`

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

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

### 6.2.8 `getDataFactory`

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

<table>
<thead>
<tr>
<th>Precondition</th>
<th>C++ component instance is running and has a <code>ComponentContext</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input Parameter</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Return</strong></td>
<td>An SDO <code>DataFactory</code> which has definitions for all complex data types used by a component.</td>
</tr>
<tr>
<td><strong>Post Condition</strong></td>
<td>An SDO <code>DataFactory</code> is constructed</td>
</tr>
</tbody>
</table>
6.2.9 getSelfReference

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

There are two variations of this API.

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

and

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

6.3 ServiceReference

The following shows the `ServiceReference` interface.

```cpp
class ServiceReference {
public:
    virtual void* getService() const = 0;
    virtual void* getCallback() const = 0;
};
```

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

6.3.1 getService

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

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

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

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

6.4 SCAException

The following shows the `SCAException` interface.

```cpp
class SCAException : public std::exception {
    public:
        const char* getEClassName() const;
        const char* getMessageText() const;
        const char* getFileName() const;
        unsigned long getLineNumber() const;
        const char* getFunctionName() const;
    };
```

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

6.4.1 getEClassName

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

<table>
<thead>
<tr>
<th>Precondition</th>
<th>C++ component instance is running and has caught an SCA Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Parameter</td>
<td></td>
</tr>
<tr>
<td>Return</td>
<td>The type of the exception as a string. e.g. “ServiceUnavailableException”</td>
</tr>
<tr>
<td>Post Condition</td>
<td>No change</td>
</tr>
</tbody>
</table>

6.4.2 getMessageText

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

<table>
<thead>
<tr>
<th>Precondition</th>
<th>C++ component instance is running and has caught an SCA Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Parameter</td>
<td></td>
</tr>
<tr>
<td>Return</td>
<td>The message which the SCA runtime attached to the exception</td>
</tr>
<tr>
<td>Post Condition</td>
<td>No change</td>
</tr>
</tbody>
</table>
6.4.3 getFileName

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

<table>
<thead>
<tr>
<th>Precondition</th>
<th>C++ component instance is running and has caught an SCA Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Parameter</td>
<td></td>
</tr>
<tr>
<td>Return</td>
<td>The filename within which the exception occurred – Will be an empty string if the filename is not known</td>
</tr>
<tr>
<td>Post Condition</td>
<td>No change</td>
</tr>
</tbody>
</table>

6.4.4 getLineNumber

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

<table>
<thead>
<tr>
<th>Precondition</th>
<th>C++ component instance is running and has caught an SCA Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Parameter</td>
<td></td>
</tr>
<tr>
<td>Return</td>
<td>The line number at which the exception occurred – Will 0 if the line number is not known</td>
</tr>
<tr>
<td>Post Condition</td>
<td>No change</td>
</tr>
</tbody>
</table>

6.4.5 getFunctionName

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

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

6.5 SCANullPointerException

The following shows the `SCANullPointerException` interface.

```cpp
class SCANullPointerException : public SCAException {
};
```

6.6 ServiceRuntimeException

The following shows the `ServiceRuntimeException` interface.

```cpp
class ServiceRuntimeException : public SCAException {
};
```
6.7 ServiceUnavailableException

The following shows the `ServiceUnavailableException` interface.

```cpp
class ServiceUnavailableException : public ServiceRuntimeException {
};
```

6.8 MultipleServicesException

The following shows the `MultipleServicesException` interface.

```cpp
class MultipleServicesException : public ServiceRuntimeException {
};
```
7  C++ Contributions

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

7.1 Executable files

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

7.1.1 Executable in contribution

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

The following shows a contribution containing a DLL.

```
META-INF/
sca-contribution.xml
bin/
  autoinsurance.dll
AutoInsurance/
  AutoInsurance.composite
  AutoInsuranceService/
    AutoInsurance.h
    AutoInsuranceImpl.componentType
  include/
    Customers.h
    Underwriting.h
    RateUtils.h
```

The SCDL for the AutoInsuranceService component is:

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

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

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

```
<contribution>
  <deployable composite="myNS:RateUtilities"
```
It is also possible to export only a subtree of a contribution. If a contribution contains the following:

```
META-INF/
  sca-contribution.xml
bin/
  rates.dll
RateUtilities/
  RateUtilities.composite
  RateUtilitiesService/
    RateUtils.h
    RateUtilsImpl.componentType
```

An export of the form:

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

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

### 7.1.3 Executable outside of contribution (Import)

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

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

The SCDL for the UnderwritingService component is:

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

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

```
<contribution>
  <deployable composite="myNS:CustomerUtilities"
    <import.cpp name="usr-bin" location="/usr/bin/" />
</contribution>
```
7.2 componentType files

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

```
META-INF/
  sca-contribution.xml
bin/
  autoinsurance.dll
AutoInsurance/
  AutoInsurance.composite
  AutoInsuranceService/
    AutoInsurance.h
    AutoInsuranceImpl.componentType
```

The SCDL for the AutoInsuranceService component is:

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

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

```
<component name="UnderwritingService">
  <implementation.cpp library="rates" path="rates:bin/"
    class="UnderwritingImpl" componentType="rates:types/UnderwritingImpl"
  />
</component>
```

7.3 C++ Contribution Extensions

7.3.1 Export.cpp

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

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

The `export.cpp` element has the following attributes:

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

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

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

The `import.cpp` element has the following attributes:

- **name : QName (1..1)** – name of the import. The `@name` attribute of a `<import.cpp/>` child element of a `<contribution/>` MUST be unique amongst the `<import.cpp/>` elements in of that contribution.

  [CPP70002]

- **location : string (1..1)** – either the QName of a export or a file system location. If the value does not match an export name it is taken as an absolute file system path.
8 Types Supported in Service Interfaces

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

8.1 Local service

For a local service the types that are supported are:

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

8.2 Remotable service

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

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

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

- Pointers.
- References.
9 Restrictions on C++ header files

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

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

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

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

- Macros
- Inline member functions
- Friend classes [CPP90002]
10 WSDL to C++ and C++ to WSDL Mapping

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

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

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

The following limitations apply:

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

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

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

Major divergences from JAX-WS:

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

10.1.1 Mapping WSDL targetNamespace to a C++ namespace

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

10.1.2 Mapping WSDL Faults to C++ Exceptions

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

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

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

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

10.1.2.1 Multiple Fault References

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

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

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

- For unwrapped messages, an SCA implementation MUST map:
  - in - the message part to a member function parameter, passed by const-reference.
  - out - the message part to a member function parameter, passed by reference, or to the member function return type, returned by-value.
  - in/out - the message part to a member function parameter, passed by reference. [CPP100003]

- For wrapped messages, an SCA implementation MUST map:
  - in - the wrapper child to a member function parameter, passed by const-reference.
  - out - the wrapper child to a member function parameter, passed by reference, or to the member function return type, returned by-value.
  - in/out - the wrapper child to a member function parameter, passed by reference. [CPP100004]
10.2 Augmentations for C++ to WSDL Mapping

10.2.1 Mapping C++ namespaces to WSDL namespaces

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

10.2.2 Parameter and return type classification

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

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

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

10.2.3 C++ to WSDL Type Conversion

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

10.2.4 Service-specific Exceptions

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

10.3 SDO Data Binding

10.3.1 Simple Content Binding

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

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

<table>
<thead>
<tr>
<th>XSD Schema Type</th>
<th>SDO Type</th>
<th>C++ Type</th>
<th>XSD Schema Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>anySimpleType</td>
<td>Object</td>
<td>UNDEFINED</td>
<td></td>
</tr>
<tr>
<td>anyType</td>
<td>DataObject</td>
<td>commonj::sdo::DataObject</td>
<td>anyType</td>
</tr>
<tr>
<td>anyURI</td>
<td>URI</td>
<td>std::string</td>
<td>string</td>
</tr>
<tr>
<td>base64Binary</td>
<td>Bytes</td>
<td>char*</td>
<td>hexBinary</td>
</tr>
<tr>
<td>Data Type</td>
<td>Implementation Type</td>
<td>C++ Type</td>
<td>Implementation Type</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------</td>
<td>----------</td>
<td>---------------------</td>
</tr>
<tr>
<td>Boolean</td>
<td></td>
<td>bool</td>
<td></td>
</tr>
<tr>
<td>Byte</td>
<td>Byte</td>
<td>int8_t</td>
<td>Byte</td>
</tr>
<tr>
<td>Date</td>
<td>YearMonthDay</td>
<td>std::string</td>
<td>string</td>
</tr>
<tr>
<td>dateTime</td>
<td>DateTime</td>
<td>std::string</td>
<td>string</td>
</tr>
<tr>
<td>Decimal</td>
<td>Decimal</td>
<td>std::string</td>
<td>UNDEFINED</td>
</tr>
<tr>
<td>Double</td>
<td>Double</td>
<td>long double</td>
<td>double</td>
</tr>
<tr>
<td>Duration</td>
<td>Duration</td>
<td>std::string</td>
<td>string</td>
</tr>
<tr>
<td>ENTITIES</td>
<td>Strings</td>
<td>std::string</td>
<td>UNDEFINED</td>
</tr>
<tr>
<td>ENTITY</td>
<td>String</td>
<td>std::string</td>
<td>string</td>
</tr>
<tr>
<td>Float</td>
<td>Float</td>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>gDay</td>
<td>Day</td>
<td>std::string</td>
<td>string</td>
</tr>
<tr>
<td>gMonth</td>
<td>Month</td>
<td>std::string</td>
<td>string</td>
</tr>
<tr>
<td>gMonthDay</td>
<td>MonthDay</td>
<td>std::string</td>
<td>string</td>
</tr>
<tr>
<td>gYear</td>
<td>Year</td>
<td>std::string</td>
<td>string</td>
</tr>
<tr>
<td>gYearMonth</td>
<td>YearMonth</td>
<td>std::string</td>
<td>string</td>
</tr>
<tr>
<td>hexBinary</td>
<td>Bytes</td>
<td>char*</td>
<td>hexBinary</td>
</tr>
<tr>
<td>ID</td>
<td>String</td>
<td>std::string</td>
<td>string</td>
</tr>
<tr>
<td>IDREF</td>
<td>String</td>
<td>std::string</td>
<td>string</td>
</tr>
<tr>
<td>IDREFS</td>
<td>Strings</td>
<td>std::string</td>
<td>UNDEFINED</td>
</tr>
<tr>
<td>Int</td>
<td>Int</td>
<td>int32_t</td>
<td>int</td>
</tr>
<tr>
<td>Integer</td>
<td>Integer</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>language</td>
<td>String</td>
<td>std::string</td>
<td>string</td>
</tr>
<tr>
<td>Long</td>
<td>Long</td>
<td>int64_t</td>
<td>long</td>
</tr>
<tr>
<td>Name</td>
<td>String</td>
<td>std::string</td>
<td>string</td>
</tr>
<tr>
<td>NCName</td>
<td>String</td>
<td>std::string</td>
<td>string</td>
</tr>
<tr>
<td>negativeInteger</td>
<td>Integer</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>NM_TOKEN</td>
<td>String</td>
<td>std::string</td>
<td>string</td>
</tr>
<tr>
<td>NM_TOKENS</td>
<td>Strings</td>
<td>std::string</td>
<td>UNDEFINED</td>
</tr>
<tr>
<td>nonNegativeInteger</td>
<td>Integer</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>nonPositiveInteger</td>
<td>Integer</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>normalizedString</td>
<td>String</td>
<td>std::string</td>
<td>string</td>
</tr>
<tr>
<td>NOTATION</td>
<td>String</td>
<td>std::string</td>
<td>string</td>
</tr>
<tr>
<td>positiveInteger</td>
<td>Integer</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1127

<table>
<thead>
<tr>
<th>QName</th>
<th>URI</th>
<th>std::string</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>Short</td>
<td>int16_t</td>
<td>short</td>
</tr>
<tr>
<td>string</td>
<td>String</td>
<td>std::string</td>
<td>string</td>
</tr>
<tr>
<td>time</td>
<td>Time</td>
<td>std::string</td>
<td>string</td>
</tr>
<tr>
<td>Token</td>
<td>String</td>
<td>std::string</td>
<td>string</td>
</tr>
<tr>
<td>unsignedByte</td>
<td>Short</td>
<td>int16_t</td>
<td>short</td>
</tr>
<tr>
<td>unsignedInt</td>
<td>Long</td>
<td>int64_t</td>
<td>long</td>
</tr>
<tr>
<td>unsignedLong</td>
<td>Integer</td>
<td>UNDEFINED</td>
<td></td>
</tr>
<tr>
<td>unsignedShort</td>
<td>Int</td>
<td>int32_t</td>
<td>int</td>
</tr>
</tbody>
</table>

1128 **10.3.2 Complex Content Binding**

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

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

11.1 Conformance Targets

The conformance targets of this specification are:

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

11.2 Conformance Claims

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

- It MUST state which conformance targets it implements.

11.3 SCA Implementations

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

1. It MUST conform to the SCA Assembly Model Specification [ASSEMBLY] and the SCA Policy Framework [POLICY].
2. It MUST implement the SCA C++ API defined in section C++ API.
3. It MUST implement the mapping between C++ and WSDL 1.1 [WSDL11] as described in WSDL to C++ and C++ to WSDL Mapping.
4. It MUST support C++ contributions as defined in C++ Contributions.

11.4 SCDL Documents

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

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

11.5 C++ Component Implementations

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

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

11.6 C++ Header Files

A C++ header file conforms to this specification if it meets the following conditions:
1. It conforms to the requirements and restrictions for a C++ header file specified in section Restrictions on C++ header files.
2. If it contains annotations, the formats and restrictions in section C++ SCA Annotations, C++ SCA Policy Annotations and C++ WSDL Mapping Annotations are followed.

11.7 WSDL Files

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

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

11.8 Extensions

[What extension points do we want to identify?].
A C++ SCA Annotations

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

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

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

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

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

- **Class**
  - The annotation immediately precedes the class.
  - Example:
    ```
    // @Scope("composite")
    class LoanServiceImpl : public LoanService {
        ...
    }; 
    ```

- **Member function**
  - The annotation immediately precedes the member function.
  - Example:
    ```
    class LoanService
    {
      public:
        // @OneWay
        virtual void reportEvent(int eventId) = 0;
        ...
    }; 
    ```

- **Data Member**
  - The annotation immediately precedes the data member.
  - Example:
    ```
    // @Property(name="loanType", type="xsd:int")
    long loanType;
    ```

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

### A.2 Interface Header Annotations

This section lists the annotations that can be used in the header file that defines a service interface.
A.2.1 @Interface

Annotation used to indicate a class defines an interface when multiple classes exist in a header file. An SCA implementation MUST treat a class with an @WebService annotation specified. [CPPA0003]

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

Format:

// @Interface

Applies to: Class

Example:

Interface header:

// @Interface
class LoanService {
...
};

Service definition:

<service name="LoanService">
  <interface.cpp header="LoanService.h" class="LoanService" />
</service>

A.2.2 @Remotable

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

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

Format:

// @Remotable

The default is false (not remotable).

Applies to: Class

Example:

Interface header:

// @Remotable
class LoanService {
...
};

Service definition:

<service name="LoanService">
  <interface.cpp header="LoanService.h" remotable="true" />
</service>
A.2.3 @Callback

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

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

Format:

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

where

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

Applies to: Class

Example:

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

Service definition:

```xml
<service name="MyService">
    <interface.cpp header="MyService.h"
    callbackHeader="MyServiceCallback.h"
    callbackClass="MyServiceCallback" />
</service>
```

A.2.4 @OneWay

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

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

Format:

```cpp
// @OneWay
```

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

Applies to: Member function

Example:

```cpp
Interface header:
class LoanService {
public:
    // @OneWay
    virtual void reportEvent(int eventId) = 0;
```
1309
1310
1311
Service definition:
1313
<service name="LoanService">
  <interface.cpp header="LoanService.h">
    <function name="reportEvent" oneWay="true" />
  </interface.cpp>
</service>

A.3 Implementation Header Annotations

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

A.3.1 @ComponentType

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

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

Format:

    // @ComponentType

Applies to: Class

Example:

Implementation header:

    // @ComponentType
    class LoanServiceImpl : public LoanService {
    ... 
    
    Component definition:

    <component name="LoanService">
      <implementation.cpp library="loan" class="LoanServiceImpl"
        class="LoanServiceImpl" />
    </component>

A.3.2 @Scope

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

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

Format:

    // @Scope("value")

where

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

Example:

Implementation header:

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

Component definition:

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

A.3.3 @EagerInit

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

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

Format:

```cpp
// @EagerInit
```

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

Applies to: Class

Example:

Implementation header:

```cpp
// @EagerInit
class LoanServiceImpl : public LoanService {
... 
};
```

Component definition:

```xml
<component name="LoanService">
  <implementation.cpp library="loan" class="LoanServiceImpl"
    eagerInit="true" />
</component>
```

A.3.4 @AllowsPassByReference

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

Corresponds to: @allowsPassByReference="true" attribute of an implementation.cpp element or a function child element of an implementation.cpp element.
Format:

```
// @AllowsPassByReference
```

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

**Applies to:** Class or Member function

**Example:**

```
Implementation header:
// @AllowsPassByReference
class LoanService {
  ...
};
```

**Component definition:**

```xml
<component name="LoanService">
  <implementation.cpp library="loan" class="LoanServiceImpl"
    allowsPassByReference="true" />
</component>
```

**A.3.5 @Property**

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

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

**Format:**

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

where

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

**Applies to:** DataMember

**Example:**

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

**Component Type definition:**

```xml
<componentType ... >
  <service ... />
</componentType>
```
A.3.6 @Reference

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

Corresponds to: reference element of a componentType element.

Format:

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

where

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

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

Applies to: Data Member

Example:

Implementation:

```
// @Reference(interfaceHeader="LoanService.h" required="true")
LoanService* loanService;
```

```
// @Reference(interfaceHeader="LoanService.h" required="false")
std::list<LoanService*> loanServices;
```

Component Type definition:

```
<componentType ... >
  <service ... />
  <reference name="loanService" multiplicity="1..1">
    <interface.cpp header="LoanService.h" class="LoanService" />
  </reference>
  <reference name="loanServices" multiplicity="0..n">
    <interface.cpp header="LoanService.h" class="LoanService" />
  </reference>
</componentType>
```

A.4 Base Annotation Grammar

```
<annotation> ::= // @<baseAnnotation>
<baseAnnotation> ::= <name> [(<params>)]
<params> ::= <paramNameValue>[, <paramNameValue>]*
```

SCA Client and Implementation Model Specification for C++ Version 1.1
Copyright © OASIS® 2006, 2009. All Rights Reserved.
• Adjacent string constants are concatenated
• **NCName** is as defined by XML schema [*XSD*]
• Whitespace including newlines between tokens is ignored.
• Annotations with parameters may span multiple lines within a comment, and are considered complete when the terminating "\" is reached.
B C++ SCA Policy Annotations

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

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

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

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

B.1 General Intent Annotations

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

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

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

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

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

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

```
// @Define SCA_PREFIX "{http://docs.oasis-open.org/ns/opencsa/sca/200712}"
// @Define CONFIDENTIALITY SCA_PREFIX ## "confidentiality"
// @Define CONFIDENTIALITY_MESSAGE CONFIDENTIALITY ## ".message"
```
Notice that, by convention, qualified intents include the qualifier as part of the name of the constant, separated by an underscore. These intent constants are defined in the file that defines an annotation for the intent (annotations for intents, and the formal definition of these constants, are covered in a following section).

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

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

Format:
```c++
// @Requires("qualifiedIntent" | {"qualifiedIntent" [, "qualifiedIntent"]})
```
where
```c++
qualifiedIntent ::= QName | QName.qualifier | QName.qualifier1.qualifier2
```

Applies to: Class, Member function, Data Member

Examples:

Attaching the intents "confidentiality.message" and "integrity.message".
```c++
// @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
```

A reference requiring support for confidentiality:
```c++
class Foo {
  ...
  // @Requires(CONFIDENTIALITY)
  // @Reference(interfaceHeader="SetBar.h")
  void setBar(Bar* bar);
  ...
}
```

Users may also choose to only use constants for the namespace part of the QName, so that they may add new intents without having to define new constants. In that case, this definition would instead look like this:
```c++
class Foo {
  ...
  // @Requires(SCA_PREFIX "confidentiality ")
  // @Reference(interfaceHeader="SetBar.h")
  void setBar(Bar* bar);
  ...
}
```

B.2 Specific Intent Annotations

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

The general form of these specific intent annotations is an annotation with a name derived from the name of the intent itself. If the intent is a qualified intent, qualifiers are supplied as an attribute to the annotation in the form of a string or an array of strings.
For example, the SCA confidentiality intent described in General Intent Annotations using the @Requires(CONFIDENTIALITY) intent can also be specified with the specific @Confidentiality intent annotation. The specific intent annotation for the "integrity" security intent is:

```
// @Integrity
```

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

**Format:**

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

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

```
Intent ::= NCName
qualifiers ::= "qualifier" | {"qualifier", "qualifier"}
qualifier ::= NCName | NCName/qualifier
```

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

**Example:**

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

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

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

### B.2.1 Security Interaction

<table>
<thead>
<tr>
<th>Intent</th>
<th>Annotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>authentication</td>
<td>@Authentication</td>
</tr>
<tr>
<td>confidentiality</td>
<td>@Confidentiality</td>
</tr>
<tr>
<td>integrity</td>
<td>@Integrity</td>
</tr>
</tbody>
</table>

These three intents can be qualified with

- transport
- message

### B.2.2 Security Implementation

<table>
<thead>
<tr>
<th>Intent</th>
<th>Annotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>runAs</td>
<td>@RunAs(role&quot;role&quot;)</td>
</tr>
<tr>
<td>Allow</td>
<td>@Allow(roles=&quot;&lt;comma separated list of roles&gt;&quot;)</td>
</tr>
<tr>
<td>permitAll</td>
<td>@PermitAll</td>
</tr>
<tr>
<td>denyAll</td>
<td>@DenyAll</td>
</tr>
</tbody>
</table>
In addition to allow roles to defined, an SCA runtime MAY use the following annotation

```
@DeclareRoles(<comma separated list of roles>)
```

### B.2.3 Reliable Messaging

<table>
<thead>
<tr>
<th>Intent</th>
<th>Annotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>atLeastOnce</td>
<td>@AtLeastOnce</td>
</tr>
<tr>
<td>atMostOnce</td>
<td>@AtMostOnce</td>
</tr>
<tr>
<td>Ordered</td>
<td>@Ordered</td>
</tr>
<tr>
<td>exactlyOnce</td>
<td>@ExactlyOnce</td>
</tr>
</tbody>
</table>

### B.2.4 Transactions

<table>
<thead>
<tr>
<th>Intent</th>
<th>Annotation</th>
<th>Qualifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>managedTransaction</td>
<td>@ManagedTransaction</td>
<td>Local</td>
</tr>
<tr>
<td></td>
<td></td>
<td>global</td>
</tr>
<tr>
<td>transactedOneWay</td>
<td>@TransactedOneWay</td>
<td></td>
</tr>
<tr>
<td>immediateOneWay</td>
<td>@ImmediateOneWay</td>
<td></td>
</tr>
<tr>
<td>propagates Transaction</td>
<td>@PropagatesTransaction</td>
<td></td>
</tr>
<tr>
<td>suspendsTransaction</td>
<td>@SuspendsTransaction</td>
<td></td>
</tr>
</tbody>
</table>

### B.2.5 Miscellaneous

<table>
<thead>
<tr>
<th>Intent</th>
<th>Annotation</th>
<th>Qualifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOAP</td>
<td>@SOAP</td>
<td>1_1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1_2</td>
</tr>
<tr>
<td>JMS</td>
<td>@JMS</td>
<td></td>
</tr>
</tbody>
</table>

### B.3 Application of Intent Annotations

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

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

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

If an annotation is specified at both the class/interface level and the member function or data member level, then the member function or data member level annotation completely overrides the class level annotation of the same type.
The intent annotation can be applied either to classes or to class member functions when adding
annotated policy on SCA services.

**B.4 Inheritance and Intent Annotations**

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

```cpp
// @Remotable
// @Integrity("transport")
// @Authentication
class HelloService {
  public:
    // @Integrity
    // @Authentication("message")
    wchar_t* hello(wchar_t* message) {...}
    // @Integrity
    // @Authentication("transport")
    wchar_t* helloThere() {...}
}

// @Remotable
// @Confidentiality("message")
class HelloChildService : public HelloService {
  public:
    // @Confidentiality("transport")
    wchar_t* hello(wchar_t* message) {...}
    // @Authentication
    wchar_t* helloWorld(){...}
}
```

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

- The effective intent annotation on the helloWorld member function is @Integrity("transport"),
  @Authentication, and @Confidentiality("message").
- The effective intent annotation on the hello member function of the HelloChildService is
  @Integrity("transport"), @Authentication, and @Confidentiality("transport"),
- The effective intent annotation on the helloThere member function of the HelloChildService is
  @Integrity and @Authentication("transport"), the same as in HelloService class.
- The effective intent annotation on the hello member function of the HelloService is @Integrity and
  @Authentication("message")

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

```xml
<?xml version="1.0" encoding="ASCII"?>
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" name="HelloServiceComposite">
  <service name="HelloService " requires="integrity/transport authentication">
    ...
  </service>
  <service name="HelloChildService" requires="integrity/transport authentication confidence/message">
```
Example 1b. Declarative intents equivalent to annotated intents in Example 1a.

B.5 Relationship of Declarative and Annotated Intents

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

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

B.6 Policy Set Annotations

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

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

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

Format:

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

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

Applies to: Class, Member function, Data Member
In this case, the Policy Sets WS_Encryption_Policy and WS_Authentication_Policy are applied, both using the namespace defined for the constant MY_NS.

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

### B.7 Policy Annotation Grammar Additions

```
<annotation> ::= // @<baseAnnotation> | @<requiresAnnotation> | @<intentAnnotation> | @<policySetAnnotation>

<requiresAnnotation> ::= Requires(<intents>)

<intents> ::= "<qualifiedIntent>" | {"<qualifiedIntent>"[ , "<qualifiedIntent>"]*}

<qualifiedIntent> ::= <intentName> | <intentName>.<qualifier> | <intentName>.<qualifier>.qualifier

<intentName> ::= {aAnyURI}NCName

<intentAnnotation> ::= <intent>[({qualifiers})]

<intent> ::= NCName [ (param) ]

<qualifiers> ::= "<qualifier>" | { "<qualifier>"[ , "<qualifier>"]* }

<qualifier> ::= NCName | NCName/<qualifier>

<policySetAnnotation> ::= policySets(<policysets>)

<policySets> ::= "<policySetName>" | {"<policySetName>"[ , "<policySetName>"]*}

<policySetName> ::= {aAnyURI}NCName

• anyURI is as defined by XML schema [XSD]
```

### B.8 Annotation Constants

```
<annotationConstant> ::= // @Define <identifier> <token string>

<identifier> ::= token

<token string> ::= "string" | "string"[ ## <token string>]

• Constants are immediately expanded
```
C++ WSDL Mapping Annotations

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

C.1 Interface Header Annotations

C.1.1 @WebService

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

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

Format:

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

where:

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

Applies to: Class

Example:

Input C++ source file:

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

```cpp
class StockQuoteService {
};
```

Generated WSDL file:

```xml
<definitions xmlns="http://schemas.xmlsoap.org/wsd/"
```
C.1.2 @WebFunction

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

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

Format:

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

where:

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

Applies to: Member function.

Example:

```
// @WebService(name="StockQuote", targetNamespace="http://www.example.org/
// serviceName="StockQuoteService")
class StockQuoteService {
  // @WebFunction(operationName="GetLastTradePrice", action="urn:GetLastTradePrice")
  float getLastTradePrice(const std::string& tickerSymbol);
  // @WebFunction(exclude=true)
  void setLastTradePrice(const std::string& tickerSymbol, float value);
};
```
Generated WSDL file:

```xml
<definitions xmlns="http://schemas.xmlsoap.org/wSDL/
xmlns:soap="http://schemas.xmlsoap.org/wSDL/soap/
xmlns:tns="http://www.example.org/"
xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
targetNamespace="http://www.example.org/">
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://www.example.org/"
    attributeFormDefault="unqualified"
    elementFormDefault="unqualified"
    targetNamespace="http://www.example.org/">
    <xs:element name="GetLastTradePrice" type="tns:GetLastTradePrice"/>
    <xs:element name="GetLastTradePriceResponse" type="tns:GetLastTradePriceResponse"/>
    <xs:complexType name="GetLastTradePrice">
      <xs:sequence>
        <xs:element name="tickerSymbol" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="GetLastTradePriceResponse">
      <xs:sequence>
        <xs:element name="return" type="xs:float"/>
      </xs:sequence>
    </xs:complexType>
  </xs:schema>
  <message name="GetLastTradePrice">
    <part name="parameters" element="tns:GetLastTradePrice"/>
  </message>
  <message name="GetLastTradePriceResponse">
    <part name="parameters" element="tns:GetLastTradePriceResponse"/>
  </message>
  <portType name="StockQuote">
    <cpp:bindings>
      <cpp:class name="StockQuoteService"/>
    </cpp:bindings>
    <operation name="GetLastTradePrice">
      <cpp:bindings>
        <cpp:memberFunction name="getLastTradePrice"/>
      </cpp:bindings>
      <input name="GetLastTradePrice" message="tns:GetLastTradePrice"/>
      <output name="GetLastTradePriceResponse" message="tns:GetLastTradePriceResponse"/>
    </operation>
  </portType>
  <binding name="StockQuoteServiceSoapBinding">
    <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="GetLastTradePrice">
      <soap:operation soapAction="urn:GetLastTradePrice" style="document"/>
      <wsdl:input name="GetLastTradePrice"/>
      <soap:body use="literal"/>
      <wsdl:input name="GetLastTradePriceResponse"/>
    </wsdl:operation>
  </binding>
</definitions>
```
C.1.3 @OneWay

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

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

Format:

```
// @OneWay
```

Applies to: Member function.

Example:

Input C++ source file:

```
// @WebService(name="StockQuote", targetNamespace="http://www.example.org/"
//     serviceName="StockQuoteService")
class StockQuoteService {
    // @WebFunction(operationName="GetLastTradePrice",
    //     action="urn:GetLastTradePrice")
    // @OneWay
    float getLastTradePrice(const std::string& tickerSymbol);
```

Generated WSDL file:

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/
xmlns:tns="http://www.example.org/
xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
targetNamespace="http://www.example.org/">
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:tns="http://www.example.org/"
        attributeFormDefault="unqualified"
        elementFormDefault="unqualified"
        targetNamespace="http://www.example.org/">
        <xs:complexType name="GetLastTradePrice">
            <xs:sequence>
                <xs:element name="tickerSymbol" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
    </xs:schema>
    <message name="GetLastTradePrice">
        <soap:body use="literal"/>
    </message>
</definitions>
```
C.1.4 @WebParam

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

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

Format:

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

where:

- **paramName : NCName (1..1)** – specifies the name of the parameter that this annotation applies to. Only named parameters MAY be referenced by an @WebParam annotation. [CPPC0004]

- **name : NCName (0..1)** – specifies the name of the associated WSDL part or element. The default value is the name of the parameter. If an @WebParam annotation is not present, and the parameter is unnamed, then a name of "argN", where N is an incrementing value from 1 indicating the position of the parameter in the argument list, will be used.

- **targetNamespace : string (0..1)** – specifies the target namespace for the part. The default namespace is the namespace of the associated @WebService. The targetNamespace attribute is ignored unless the binding style is document, and the binding parameterStyle is bare. See @SOAPBinding.
- **mode**: token (0..1) – specifies whether the parameter is associated with the input message, output message, or both. The default value is determined by the passing mechanism for the parameter, see Parameter and return type classification.

- **header**: boolean (0..1) – specifies whether this parameter is associated with a SOAP header element. The default value is “false”.

- **partName**: NCName (0..1) – specifies the name of the WSDL part associated with this item. The default value is the value of name.

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

### Applies to:
Member function parameter.

### Example:

**Input C++ source file:**

```cpp
// @WebService(name="StockQuote", targetNamespace="http://www.example.org/
// serviceName="StockQuoteService")
class StockQuoteService {
    // @WebFunction(operationName="GetLastTradePrice",
    // action="urn:GetLastTradePrice")
    // @WebParam(paramName="tickerSymbol", name="symbol")
    float getLastTradePrice(const std::string& tickerSymbol);
};
```

**Generated WSDL file:**

```xml
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/
xmlns:tns="http://www.example.org/
xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
targetNamespace="http://www.example.org/">
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:tns="http://www.example.org/
        attributeFormDefault="unqualified"
        elementFormDefault="unqualified"
        targetNamespace="http://www.example.org/">
        <xs:element name="GetLastTradePrice" type="tns:GetLastTradePrice"/>
        <xs:element name="GetLastTradePriceResponse"
            type="tns:GetLastTradePriceResponse"/>
        <xs:complexType name="GetLastTradePrice">
            <xs:sequence>
                <xs:element name="symbol" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
        <xs:complexType name="GetLastTradePriceResponse">
            <xs:sequence>
                <xs:element name="return" type="xs:float"/>
            </xs:sequence>
        </xs:complexType>
    </xs:schema>
    <message name="GetLastTradePrice">
        <part name="parameters" element="tns:GetLastTradePrice"/>
    </message>
</definitions>
```
<message name="GetLastTradePriceResponse">
  <part name="parameters" element="tns:GetLastTradePriceResponse"/>
</message>

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

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

<service name="StockQuoteService">
  <port name="StockQuotePort" binding="tns:StockQuoteServiceSoapBinding">
    <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
  </port>
</service>
</definitions>

C.1.5 @WebResult

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

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

Format:

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

where:

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

---

SCA Client and Implementation Model Specification for C++ Version 1.1
Copyright © OASIS® 2006, 2009. All Rights Reserved.
• **targetNamespace : string (0..1)** – specifies the target namespace for the part. The default namespace is the namespace of the associated @WebService. The targetNamespace attribute is ignored unless the binding style is document, and the binding parameterStyle is bare. See @SOAPBinding.

• **header : boolean (0..1)** – specifies whether the result is associated with a SOAP header element. The default value is "false".

• **partName : NCName (0..1)** – specifies the name of the WSDL part associated with this item. The default value is the value of name.

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

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

**Example:**

Input C++ source file:

```cpp
// @WebService(name="StockQuote", targetNamespace="http://www.example.org/
//       serviceName="StockQuoteService")
class StockQuoteService {
    // @WebFunction(operationName="GetLastTradePrice",
    //       action="urn:GetLastTradePrice")
    // @WebResult(name="price")
    float getLastTradePrice(const std::string& tickerSymbol);
};
```

Generated WSDL file:

```xml
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/
   xmlns:tns="http://www.example.org/
   xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
   targetNamespace="http://www.example.org/">
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
             xmlns:tns="http://www.example.org/
             attributeFormDefault="unqualified"
             elementFormDefault="unqualified"
             targetNamespace="http://www.example.org/">
    <xs:element name="GetLastTradePrice" type="tns:GetLastTradePrice"/>
    <xs:element name="GetLastTradePriceResponse" type="tns:GetLastTradePriceResponse"/>
    <xs:complexType name="GetLastTradePrice">
        <xs:sequence>
            <xs:element name="tickerSymbol" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="GetLastTradePriceResponse">
        <xs:sequence>
            <xs:element name="price" type="xs:float"/>
        </xs:sequence>
    </xs:complexType>
  </xs:schema>
  <message name="GetLastTradePrice">
    <part name="parameters" element="tns:GetLastTradePrice"/>
  </message>
</definitions>
```
<message name="GetLastTradePriceResponse">
  <part name="parameters" element="tns:GetLastTradePriceResponse"/>
</message>

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

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

<service name="StockQuoteService">
  <port name="StockQuotePort" binding="tns:StockQuoteServiceSoapBinding">
    <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
  </port>
</service>
</definitions>

C.1.6 @SOAPBinding

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

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

Format:

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

where:

- **style : token (0..1)** – specifies the WSDL binding style. The default value is "DOCUMENT".
- **use : token (0..1)** – specifies the WSDL binding use. The default value is "LITERAL".
- **parameterStyle : token (0..1)** – specifies the WSDL parameter style. The default value is "WRAPPED".
**Applies to:** Class, Member function.

**Example:**

Input C++ source file:

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

Generated WSDL file:

```xml
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/
    xmlns:tns="http://www.example.org/
    xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
    targetNamespace="http://www.example.org/">
  <portType name="StockQuote">
    <cpp:bindings>
      <cpp:class name="StockQuoteService"/>
    </cpp:bindings>
  </portType>
  <binding name="StockQuoteServiceSoapBinding">
    <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http"/>
  </binding>
  <service name="StockQuoteService">
    <port name="StockQuotePort" binding="tns:StockQuoteServiceSoapBinding">
      <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
    </port>
  </service>
</definitions>
```

**C.1.7 @WebFault**

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

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

**Format:**

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

where:

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

**Applies to:** Class.
Example:

Input C++ source file:

```c++
// @WebFault(name="UnknownSymbolFault",
//     targetNamespace="http://www.example.org/")
class UnknownSymbol {
    UnknownSymbol(const char* message,
        const std::string& faultInfo);
    std::string getFaultInfo() const;
};

// @WebService(name="StockQuote", targetNamespace="http://www.example.org/
//       serviceName="StockQuoteService")
class StockQuoteService {
    // @WebFunction(operationName="GetLastTradePrice",
    //       action="urn:GetLastTradePrice")
    // @WebThrows(faults="UnknownSymbol")
    float getLastTradePrice(const std::string& tickerSymbol);
};
```

Generated WSDL file:

```xml
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/
    xmlns:tns="http://www.example.org/"
    xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
    targetNamespace="http://www.example.org/">
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:tns="http://www.example.org/
        attributeFormDefault="unqualified"
        elementFormDefault="unqualified"
        targetNamespace="http://www.example.org/">
        <xs:element name="GetLastTradePrice" type="tns:GetLastTradePrice"/>
        <xs:element name="GetLastTradePriceResponse" type="tns:GetLastTradePriceResponse"/>
        <xs:complexType name="GetLastTradePrice">
            <xs:sequence>
                <xs:element name="tickerSymbol" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
        <xs:complexType name="GetLastTradePriceResponse">
            <xs:sequence>
                <xs:element name="return" type="xs:float"/>
            </xs:sequence>
        </xs:complexType>
        <xs:element name="UnknownSymbolFault" type="xs:string"/>
    </xs:schema>
    <message name="GetLastTradePrice">
        <part name="parameters" element="tns:GetLastTradePrice"/>
    </message>
    <message name="GetLastTradePriceResponse">
        <part name="parameters" element="tns:GetLastTradePriceResponse"/>
    </message>
    <message name="UnknownSymbol">
```
C.1.8 @WebThrows

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

Corresponds to: No equivalent in JAX-WS.

Format:

// @WebThrows(faults="faultMsg1", "faultMsgn")

where:

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

Example:

See @WebFault.
D  WSDL C++ Mapping Extensions

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

D.1 <cpp:bindings>
<cpp:bindings> is a container type which may be used as a WSDL extension. All other SCA wsdl extensions will be specified as children of a <cpp:bindings> element. A <cpp:bindings> element may be used as an extension to any WSDL type that accepts extensions.

D.2 <cpp:class>
<cpp:class> provides a mechanism for defining an alternate C++ class name for a WSDL construct.

Format:

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

where:

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

Applicable WSDL element(s):

- wsdl:portType
- wsdl:fault

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

Example:

Input WSDL file:

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

Generated C++ file:

```
// @WebService(name="StockQuote", targetNamespace="http://www.example.org/"
//   serviceName="StockQuoteService")
```
D.3 <cpp:enableWrapperStyle>

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

Format:

<cpp:enableWrapperStyle>value</cpp:enableWrapperStyle>

where:

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

Applicable WSDL element(s):

- **wsdl:definitions**
- **wsdl:portType** – overrides a binding applied to wsdl:definitions
- **wsdl:portType/wsdl:operation** – overrides a binding applied to wsdl:definitions or the enclosing wsdl:portType

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

Example:

Input WSDL file:

```xml
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/
 xmlns:tns="http://www.example.org/
 xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
 targetNamespace="http://www.example.org/">
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://www.example.org/
    attributeFormDefault="unqualified"
    elementFormDefault="unqualified"
    targetNamespace="http://www.example.org/">
    <xs:element name="GetLastTradePrice" type="tns:GetLastTradePrice"/>
    <xs:element name="GetLastTradePriceResponse" type="tns:GetLastTradePriceResponse"/>
    <xs:complexType name="GetLastTradePrice">
      <xs:sequence>
        <xs:element name="tickerSymbol" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="GetLastTradePriceResponse">
      <xs:sequence>
        <xs:element name="return" type="xs:float"/>
      </xs:sequence>
    </xs:complexType>
  </xs:schema>
  < message name="GetLastTradePrice">
    <part name="parameters" element="tns:GetLastTradePrice"/>
  </message>
</definitions>
```
<message name="GetLastTradePriceResponse">
    <part name="parameters" element="tns:GetLastTradePriceResponse"/>
</message>

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

Generated C++ file:

```cpp
// @WebService(name="StockQuote", targetNamespace="http://www.example.org/
//       serviceName="StockQuoteService")
class StockQuoteService {
    // @WebFunction(operationName="GetLastTradePrice",
    //       action="urn:GetLastTradePrice")
    commonj::sdo::DataObjectPtr getLastTradePrice(commonj::sdo::DataObjectPtr parameters);
};
```

D.4 <cpp:namespace>

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

**Format:**

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

where:

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

**Applicable WSDL element(s):**

- wsdl:definitions

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

[CPPD0004]

**Example:**

...
D.5 <cpp:memberFunction>

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

Format:

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

where:

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

Applicable WSDL element(s):

- `wsdl:portType/wsdl:operation`

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

Example:

Input WSDL file:

```xml
<definitions xmlns="http://schemas.xmlsoap.org/wsd1/
  xmlns:soap="http://schemas.xmlsoap.org/wsd1/soap/
  xmlns:tns="http://www.example.org/
  xmlns:cpp="http://docs.oasis-open.org/ns/opencsa/sca-c-cpp/cpp/200901"
  targetNamespace="http://www.example.org/">
  <cpp:bindings>
    <cpp:namespace name="stock"/>
  </cpp:bindings>
  <portType name="StockQuote">
    <cpp:bindings>
      <cpp:class name="StockQuoteService"/>
    </cpp:bindings>
  </portType>
</definitions>
```

Generated C++ file:

```cpp
namespace stock {
  // @WebService(name="StockQuote", targetNamespace="http://www.example.org/
  //       serviceName="StockQuoteService")
  class StockQuoteService {
  }
}
```
<xs:schema xmlns:tns="http://www.example.org/"
    attributeFormDefault="unqualified"
    elementFormDefault="unqualified"
    targetNamespace="http://www.example.org/">

    <xs:element name="GetLastTradePrice" type="tns:GetLastTradePrice"/>
    <xs:element name="GetLastTradePriceResponse" type="tns:GetLastTradePriceResponse"/>
    <xs:complexType name="GetLastTradePrice">
        <xs:sequence>
            <xs:element name="tickerSymbol" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="GetLastTradePriceResponse">
        <xs:sequence>
            <xs:element name="return" type="xs:float"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>

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

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

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

Generated C++ file:

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

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

D.6 <cpp:parameter>

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

```xml
cpp:parameter name="CPPParameter" part="WSDLPart"
    childElementName="WSDLElement" type="CPPType"/>
```

where:

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

Applicable WSDL element(s):

- `wsdl:portType/wsdl:operation`

Example:

Input WSDL file:

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

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

    <message name="GetLastTradePrice">
        <part name="parameters" element="tns:GetLastTradePrice"/>
    </message>
    <message name="GetLastTradePriceResponse">
        <part name="parameters" element="tns:GetLastTradePriceResponse"/>
    </message>
</definitions>
```
<portType name="StockQuote">
  <cpp:bindings>
    <cpp:class name="StockQuoteService"/>
  </cpp:bindings>
  <operation name="GetLastTradePrice">
    <cpp:bindings>
      <cpp:memberFunction name="getLastTradePrice"/>
      <cpp:parameter name="tickerSymbol"
        part="tns:GetLastTradePrice/parameter"
        childElementName="symbol"/>
    </cpp:bindings>
    <input name="GetLastTradePrice" message="tns:GetLastTradePrice">
    </input>
    <output name="GetLastTradePriceResponse" message="tns:GetLastTradePriceResponse">
    </output>
  </operation>
</portType>

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

<service name="StockQuoteService">
  <port name="StockQuotePort" binding="tns:StockQuoteServiceSoapBinding">
    <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
  </port>
</service>
</definitions>

Generated C++ file:

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

D.7 JAX-WS WSDL Extensions

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

[CPPD0007] The following is a list of JAX-WS WSDL extensions that MAY be recognized, and their
corresponding SCA WSDL extension.
D.8 WSDL Extensions Schema

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


The following copy is provided for reference.

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

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

elementFormDefault="qualified">
  <element name="bindings" type="cpp:BindingsType" />
  <complexType name="BindingsType">
    <choice minOccurs="0" maxOccurs="unbounded">
      <element ref="cpp:namespace" />
      <element ref="cpp:class" />
      <element ref="cpp:enableWrapperStyle" />
      <element ref="cpp:memberFunction" />
      <element ref="cpp:parameter" />
    </choice>
  </complexType>
  <element name="namespace" type="cpp:NamespaceType" />
  <complexType name="NamespaceType">
    <attribute name="name" type="xsd:anyURI" use="required" />
  </complexType>
  <element name="class" type="cpp:ClassType" />
  <complexType name="ClassType">
    <attribute name="name" type="xsd:NCName" use="required" />
  </complexType>
  <element name="memberFunction" type="cpp:MemberFunctionType" />
  <complexType name="MemberFunctionType">
    <attribute name="name" type="xsd:NCName" use="required" />
  </complexType>
  <element name="parameter" type="cpp:ParameterType" />
  <complexType name="ParameterType">
    <attribute name="part" type="xsd:string" use="required" />
    <attribute name="childElementName" type="xsd:QName" use="required" />
    <attribute name="name" type="xsd:NCName" use="required" />
    <attribute name="type" type="xsd:string" use="optional" />
  </complexType>
</schema>
```
</complexType>

<element name="enableWrapperStyle" type="xsd:boolean" />
</schema>
E XML Schemas

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

The normative schemas are located at:


The following copies are provided for reference.

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

```xml
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
    xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
    elementFormDefault="qualified">
    <include schemaLocation="sca-core.xsd"/>
    <element name="interface.cpp" type="sca:CPPInterface"
        substitutionGroup="sca:interface"/>
    <complexType name="CPPInterface">
        <complexContent>
            <extension base="sca:Interface">
                <sequence>
                    <element name="function" type="sca:CPPFunction"
                        minOccurs="0" maxOccurs="unbounded" />
                    <element name="callbackFunction" type="sca:CPPFunction"
                        minOccurs="0" maxOccurs="unbounded" />
                    <any namespace="##other" processContents="lax"
                        minOccurs="0" maxOccurs="unbounded"/>
                </sequence>
                <attribute name="header" type="string" use="required"/>
                <attribute name="class" type="Name" use="required"/>
                <attribute name="callbackHeader" type="string" use="optional"/>
                <attribute name="callbackClass" type="Name" use="optional"/>
                <attribute name="remotable" type="boolean" use="optional"/>
                <anyAttribute namespace="##other" processContents="lax"/>
            </extension>
        </complexContent>
    </complexType>
</schema>
```

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

```xml
<?xml version="1.0" encoding="UTF-8"?>
<complexType name="CPPFunction">
    <attribute name="name" type="NCName" use="required"/>
    <attribute name="oneWay" type="boolean" use="optional"/>
    <anyAttribute namespace="##other" processContents="lax"/>
</complexType>
```
E.3 sca-contribution-cpp-1.1.xsd

```xml
<schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
    xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
    elementFormDefault="qualified">

  <include schemaLocation="sca-contributions.xsd" />

  <element name="export.cpp" type="sca:CPPExport"
    substitutionGroup="sca:Export" />

  <complexType name="CPPExport">
    <sequence>
      <element name="function" type="sca:CPPImplementationFunction"
        minOccurs="0" maxOccurs="unbounded" />
      <any namespace="##other" processContents="lax"
        minOccurs="0" maxOccurs="unbounded" />
    </sequence>
    <attribute name="library" type="NCName" use="required" />
    <attribute name="header" type="NCName" use="required" />
    <attribute name="path" type="string" use="optional" />
    <attribute name="class" type="Name" use="optional" />
    <attribute name="componentType" type="string" use="optional" />
    <attribute name="scope" type="sca:CPPImplementationScope"
      use="optional" />
    <attribute name="eagerInit" type="boolean" use="optional" />
    <attribute name="allowsPassByReference" type="boolean"
      use="optional" />
    <anyAttribute namespace="##other" processContents="lax" />
  </complexType>

  <complexType name="CPPImplementationScope">
    <restriction base="string">
      <enumeration value="stateless"/>
      <enumeration value="composite"/>
    </restriction>
  </complexType>

  <complexType name="CPPImplementationFunction">
    <attribute name="name" type="NCName" use="required" />
    <attribute name="allowsPassByReference" type="boolean"
      use="optional" />
    <anyAttribute namespace="##other" processContents="lax" />
  </complexType>

</schema>
```
<complexContent>
   <attribute name="name" type="QName" use="required"/>
   <attribute name="path" type="string" use="optional"/>
</complexContent>
</complexType>
<element name="import.cpp" type="sca:CPPImport"
   substitutionGroup="sca:Import"/>
<complexType name="CPPImport">
   <complexContent>
      <attribute name="name" type="QName" use="required"/>
      <attribute name="location" type="string" use="required"/>
   </complexContent>
</complexType>
</schema>
This section contains a list of conformance items for the SCA C++ Client and Implementation Model specification.

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

| [CPP70002] | The `@name` attribute of a `<import.cpp/>` child element of a `<contribution/>` MUST be unique amongst the `<import.cpp/>` elements in of that contribution. |
| [CPP90001] | A C++ header file used to define an interface MUST:  
  - Declare at least one class with:  
    - At least one public member function.  
    - All public member functions must be pure virtual (virtual with no implementation) |
| [CPP90002] | A C++ header file used to define an interface MUST NOT use the following constructs:  
  - Macros  
  - Inline member functions  
  - Friend classes |
| [CPP100001] | A WSDL file might define a namespace using the `<sca:namespace> WSDL extension, otherwise all C++ classes MUST be placed in a default namespace as determined by the implementation. Implementations SHOULD provide a mechanism for overriding the default namespace. |
| [CPP100002] | If multiple operations within the same portType indicate that they throw faults that reference the same global element, an SCA implementation MUST generate a single C++ exception class with each C++ member function referencing this class in its `@WebThrows` annotation. |
| [CPP100003] | For unwrapped messages, an SCA implementation MUST map:  
  - `in` - the message part to a member function parameter, passed by const-reference.  
  - `out` - the message part to a member function parameter, passed by reference, or to the member function return type, returned by-value.  
  - `in/out` - the message part to a member function parameter, passed by reference. |
| [CPP100004] | For wrapped messages, an SCA implementation MUST map:  
  - `in` - the wrapper child to a member function parameter, passed by const-reference.  
  - `out` - the wrapper child to a member function parameter, passed by reference, or to the member function return type, returned by-value.  
  - `in/out` - the wrapper child to a member function parameter, passed by reference. |
| [CPP100005] | An SCA implementation SHOULD provide a mechanism for overriding the default targetNamespace. |
| [CPP100006] | An SCA implementation MUST map a method’s return type as an `out` parameter, a parameter passed by-reference or by-pointer as an `in/out` parameter, and all other parameters, including those passed by-const-reference as `in` parameters. |
| [CPP100007] | An SCA implementation must ensure each class that is referenced from an `@WebThrows` annotation MUST itself have a `@WebFault` annotation that associates the fault with a particular global element that will be associated with the fault message. |
An SCA implementation MAY support source file annotations. If annotations are supported by an implementation, the annotations defined here MUST be supported and MUST be mapped to SCDL as described. The SCA runtime MUST only process the SCDL files and not the annotations.

If multiple annotations apply to a program element, all of the annotations SHOULD be in the same comment block.

An SCA implementation MUST treat a class with an @WebService annotation specified.

An SCA implementation MAY support source file annotations for WSDL. If annotations are supported by an implementation, the annotations defined here MUST be supported and MUST be mapped to WSDL as described.

An SCA implementation MUST treat any instance of a @Interface annotation and without an explicit @WebService annotation as if a @WebService annotation with no parameters was specified.

If a @WebService does not have a **portName** element, an SCA implementation MUST use the value associated with the **name** element, suffixed with “Port”.

Only named parameters MAY be referenced by an @WebParam annotation.

The value of the type property of a @WebParam annotation MUST be one of the simpleTypes defined in namespace `http://www.w3.org/2001/XMLSchema`.

The value of the type property of a @WebResult annotation MUST be one of the simpleTypes defined in namespace `http://www.w3.org/2001/XMLSchema`.

A C++ class with a @WebFault annotation MUST provide a constructor that takes two parameters, a std::string and a type representing the fault information. Additionally, the class MUST provide a const member function “getFaultInfo” that takes no parameters, and returns the same type as defined in the constructor.

A C++ class that is listed in a @WebThrows annotation MUST itself have a @WebFault annotation.

An SCA implementation MAY support these WSDL extensions. If these extensions are supported by an implementation, all the extensions defined here MUST be supported and MUST be mapped to C++ as described.

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

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

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

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

The `@type` attribute of a `<parameter/>` element MUST be a valid C++ type.

An SCA implementation MAY support the reading and interpretation of JAX-WS defined WSDL extensions; however it MUST give precedence to the corresponding SCA WSDL extension if present.
### F.1 JAX-WS Conformance

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

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

References to jaxws:enableWrapperStyle in the conformance statement should be treated as the WSDL extension cpp:enableWrapperStyle.

2.3.1.2 Default mapping mode

2.3.1.2 Disabling wrapper style

References to jaxws:enableWrapperStyle in the conformance statement should be treated as the C++ annotation @WebResult.

2.3.1.2 Wrapped parameter naming

2.3.1.2 Parameter name clash

2.5 javax.xml.ws.WebFault required

References to javax.jws.WebFault in the conformance statement should be treated as the C++ annotation @WebFault.

2.5 Exception naming

2.5 Fault equivalence

2.6 Required WSDL extensions

2.6.1 Unbound message parts

2.6.2.1 Duplicate headers in binding

2.6.2.1 Duplicate headers in message

3 WSDL 1.1 support

3 Standard annotations

[CPPC0001]

3.1 Java identifier mapping

3.1.1 Method name disambiguation

References to javax.jws.WebMethod in the conformance statement should be treated as the C++ annotation @WebFunction.

3.2 WSDL and XML Schema import directives

3.4 portType naming

3.5 Operation naming

3.5.1 One-way mapping

References to javax.jws.OneWay in the conformance statement should be treated as the C++ annotation @WebFunction.
as the C++ annotation @OneWay.

3.5.1 One-way mapping errors

3.6.1 Parameter classification

3.6.1 Parameter naming

3.6.1 Result naming

3.6.1 Header mapping of parameters and results

References to javax.jws.WebParam in the conformance statement should be treated as the C++ annotation @WebParam.

References to javax.jws.WebResult in the conformance statement should be treated as the C++ annotation @WebResult.

3.7 Exception naming

References to javax.jws.WebFault in the conformance statement should be treated as the C++ annotation @WebFault.

3.8 Binding selection

References to the BindingType annotation should be treated as references to SOAP related intents defined by [POLICY].

3.10 SOAP binding support

3.10.1 SOAP binding style required

3.11 Port selection

3.11 Port binding

References to the BindingType annotation should be treated as references to SOAP related intents defined by [POLICY].

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

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

F.1.1 Ignored Conformance Statements

<table>
<thead>
<tr>
<th>Section</th>
<th>Conformance Statement</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Definitions mapping</td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>javax.xml.bind.XmlSeeAlso required</td>
<td></td>
</tr>
<tr>
<td>2.3.1</td>
<td>use of JAXB annotations</td>
<td></td>
</tr>
<tr>
<td>2.3.1.2</td>
<td>Using javax.xml.ws.RequestWrapper</td>
<td></td>
</tr>
<tr>
<td>2.3.1.2</td>
<td>Using javax.xml.ws.ResponseWrapper</td>
<td></td>
</tr>
<tr>
<td>2.3.3</td>
<td>Use of Holder</td>
<td></td>
</tr>
<tr>
<td>2.3.4</td>
<td>Asynchronous mapping required</td>
<td></td>
</tr>
<tr>
<td>2.3.4</td>
<td>Asynchronous mapping option</td>
<td></td>
</tr>
<tr>
<td>2.3.4.2</td>
<td>Asynchronous method naming</td>
<td></td>
</tr>
<tr>
<td>Section</td>
<td>Topic</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>2.3.4.2</td>
<td>Asynchronous parameter naming</td>
<td></td>
</tr>
<tr>
<td>2.3.4.2</td>
<td>Failed method invocation</td>
<td></td>
</tr>
<tr>
<td>2.3.4.4</td>
<td>Response bean naming</td>
<td></td>
</tr>
<tr>
<td>2.3.4.5</td>
<td>Asynchronous fault reporting</td>
<td></td>
</tr>
<tr>
<td>2.3.4.5</td>
<td>Asynchronous fault cause</td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td>JAXB class mapping</td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td>JAXB customization use</td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td>JAXB customization clash</td>
<td></td>
</tr>
<tr>
<td>2.4.1</td>
<td>javax.xml.ws.wsaddressing.W3CEndpointReference</td>
<td></td>
</tr>
<tr>
<td>2.6.3.1</td>
<td>Use of MIME type information</td>
<td></td>
</tr>
<tr>
<td>2.6.3.1</td>
<td>MIME type mismatch</td>
<td></td>
</tr>
<tr>
<td>2.6.3.1</td>
<td>MIME part identification</td>
<td></td>
</tr>
<tr>
<td>2.7</td>
<td>Service superclass required</td>
<td></td>
</tr>
<tr>
<td>2.7</td>
<td>Service class naming</td>
<td></td>
</tr>
<tr>
<td>2.7</td>
<td>javax.xml.ws.WebServiceClient required</td>
<td></td>
</tr>
<tr>
<td>2.7</td>
<td>Default constructor required</td>
<td></td>
</tr>
<tr>
<td>2.7</td>
<td>2 argument constructor required</td>
<td></td>
</tr>
<tr>
<td>2.7</td>
<td>Failed getPort Method</td>
<td></td>
</tr>
<tr>
<td>2.7</td>
<td>javax.xml.ws.WebEndpoint required</td>
<td></td>
</tr>
<tr>
<td>3.2</td>
<td>Package name mapping</td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Class mapping</td>
<td></td>
</tr>
<tr>
<td>3.4.1</td>
<td>Inheritance flattening</td>
<td></td>
</tr>
<tr>
<td>3.4.1</td>
<td>Inherited interface mapping</td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>use of JAXB annotations</td>
<td></td>
</tr>
<tr>
<td>3.6.2.1</td>
<td>Default wrapper bean names</td>
<td></td>
</tr>
<tr>
<td>3.6.2.1</td>
<td>Default wrapper bean package</td>
<td></td>
</tr>
<tr>
<td>3.6.2.3</td>
<td>Null Values in rpc/literal</td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>java.lang.RuntimeExceptions and java.rmi.RemoteExceptions</td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>Fault bean name clash</td>
<td></td>
</tr>
<tr>
<td>3.11</td>
<td>Service creation</td>
<td></td>
</tr>
</tbody>
</table>
G Migration

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

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

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

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

Participants:
Andrew Borley, IBM
Bryan Aupperle, IBM
David Haney, Rogue Wave Software
Jeff Mischkinsky, Oracle
Mike Edwards, IBM
Pete Robbins, IBM
I Non-Normative Text
**J Revision History**

[optional; should not be included in OASIS Standards]

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Editor</th>
<th>Changes Made</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>•</td>
</tr>
</tbody>
</table>