# SCA Service Component Architecture

## Client and Implementation Model Specification for C++

SCA Version 1.00, March 21 2007



| Technical Contacts: | Andrew Borley | IBM Corporation |
| --- | --- | --- |
| | David Haney | Rogue Wave Software |
| | Oisin Hurley | IONA Technologies |
| | Todd Little | BEA Systems, Inc. |
| | Brian Lorenz | Sybase, Inc. |
| | Conor Patten | IONA Technologies |
| | Pete Robbins | IBM Corporation |
| | Colin Thorne | IBM Corporation |

## Copyright Notice



## License

## Status of this Document

This specification may change before final release and you are cautioned against relying on the content of this specification. The authors are currently soliciting your contributions and suggestions. Licenses are available for the purposes of feedback and (optionally) for implementation.

# Table of Contents

# 1. Client and Implementation Model

## 1.1. Introduction

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

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

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

### 1.1.1. Use of Annotations

This document defines interface and implementation meta-data using annotations. The annotations are defined as C++ comments in interface and implementation header files, for example:

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

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

## 1.2. 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.

### 1.2.1. Implementing a Service

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

The services provided by the C++ implementation have an interface which is defined using the declaration of a C++ abstract base class. An abstract base class is a class which has only pure virtual methods. This is the service interface.

The C++ class based component implementation must implement the C++ abstract base class that defines the service interface.

The abstract base class for the service interface could be generated from a WSDL portType using the mapping defined in this specification.

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

Service interface.

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

Implementation declaration header file.

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

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

Implementation.

```
#include "LoanServiceImpl.h"

LoanServiceImpl::LoanServiceImpl()
{
      …
}
LoanServiceImpl::~LoanServiceImpl()
{
```

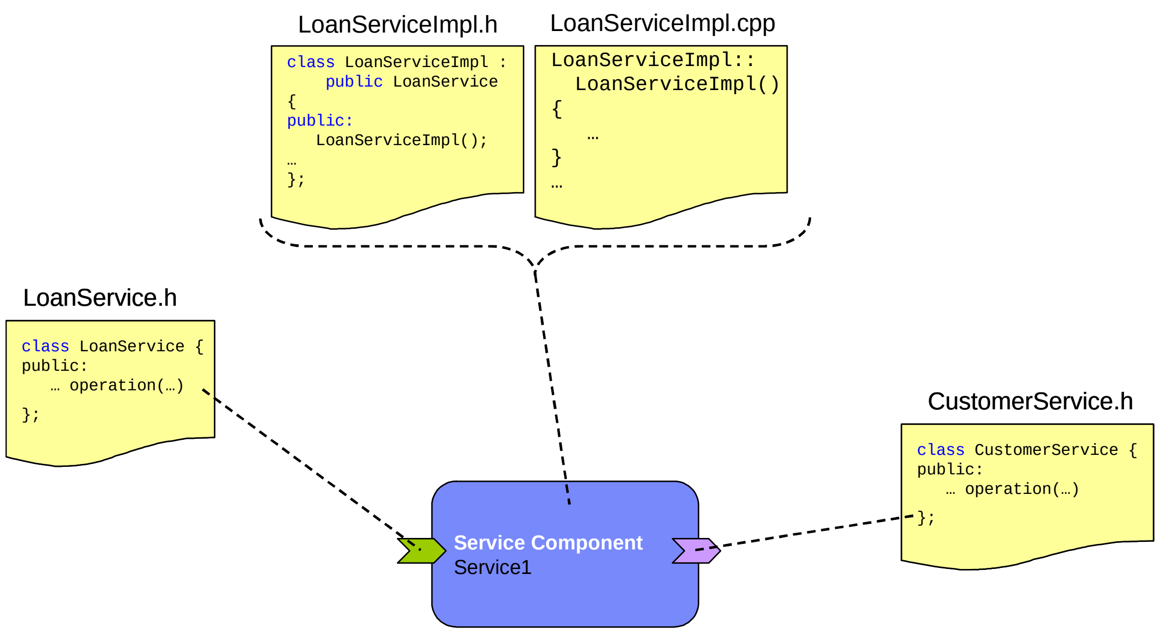
```
        …
}

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

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

```
<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0">
      <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.



#### 1.2.1.1. Implementing a Remotable Service

Remotable services are services that can be published through entry points. Published services can be accessed by clients outside of the composite that contains the component that provides the service.

Whether a service is remotable is defined by the interface of the service. This can be achieved by using the ***@Remotable*** annotation in the C++ interface header or by adding the ***remotable*** attribute to the C++ interface definition in the componentType file. The following snippet shows the annotation of the interface header:

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

The following snippet shows the component type for a remotable service:

```
<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0">
      <service name="LoanService">
            <interface.cpp header="LoanService.h" remotable="true"/>
      </service>
</componentType>
```

The style of remotable interfaces is typically ***coarse grained*** and intended for ***loosely coupled*** interactions. Remotable service Interfaces are not allowed to make use of method ***overloading***.

Complex data types exchanged via remotable service interfaces must be compatible with the marshalling technology that is used by the service binding.  For example, if the service is going to be exposed using the standard web service binding, then the parameters must be Service Data Objects (SDOs) [1].

Independent of whether the remotable service is called from outside a composite, or from another component in the same composite, the data exchange semantics are ***by-value***.

Implementations of remotable services may modify input data during or after an invocation and may modify return data after the invocation. If a remotable service is called locally or remotely, the SCA container is responsible for making sure that no modification of input data or post-invocation modifications to return data are seen by the caller.

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. The implementation of remotable services that allow pass by reference must not alter its input data during or after the invocation, and must not modify return data after invocation. The ***@AllowsPassByReference*** annotation on the implementation header of a remotable service is used to either declare that calls to the whole interface or individual methods allow pass by reference. Alternatively this can be specified in SCDL using the **allowsPassByReference=true** attribute on the implementation.cpp element or method definition.

#### *1.2.1.2. Implementing a Local Service*

A local service can only be called by clients that are part of the same composite as the component that implements the local service. Local services cannot be published through entry points.

Whether a service is local is defined by its interface. In C++ this is indicated by the service not being defined as ***remotable***.

The style of local interfaces is typically ***fine grained*** and intended for ***tightly coupled*** interactions.

The data exchange semantics for calls to local services is ***by-reference***. This means that code must be written with the knowledge that changes made to parameters (other than simple types) by either the client or the provider of the service can be seen by the other.

### 1.2.2. Conversational and Non-Conversational services

Service interfaces may be annotated to specify whether their contract is conversational, as described in the Assembly Specification [2] using the @Conversation annotation.

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

### 1.2.3. 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 may be specified using the **scope** attribute of the implementation.cpp or by specifying the ***@Scope*** annotation on the service class in the implementation header.

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

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

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

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

The following snippet shows how to define the implementation scope using the @Scope annotation:

```
// LoanServiceImpl header
// @Scope(“composite”)
class LoanServiceImpl : public LoanService
{
public:
      LoanServiceImpl();
      virtual ~LoanServiceImpl();

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

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

The Following sections specify the mandatory scopes an SCA runtime must support for C++ component implementations.

#### *1.2.3.1. Stateless scope*

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

#### *1.2.3.2. Request scope*

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

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

#### *1.2.3.3. 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 @EagerInit annotation or the eagerInit=true attribute on the 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.

#### *1.2.3.4. Conversation scope*

A conversation is defined as a series of correlated interactions between a client and a target service. A conversational scope starts when the first service request is dispatched to an implementation instance offering the target service. A conversational scope completes after an end operation defined by the service contract is called and completes processing or the conversation expires (see 1.5.3 Methods that End the Conversation). A conversation may be long-running and the SCA runtime may choose to passivate implementation instances. If this occurs, the runtime must guarantee implementation instance state is preserved.

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

### 1.2.4. 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 or by using the @Property annotation on a data member of the implementation interface class. The C++ component can retrieve the properties using the getProperties() on the ComponentContext class.

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

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

void clientMethod()
{
      …

      ComponentContext context = ComponentContext::getCurrent();

      DataObjectPtr properties = context.getProperties();

      long loanRating = properties->getInteger(“maxLoanValue”);
```

```
        …
}
```

### 1.2.5. Component Type and Component

For a C++ component implementation, a component type must be specified in a side file. The component type side file must be located in the same composite directory as the header file for the implementation class.

This Client and Implementation Model for C++ extends the SCA Assembly model [2] providing support for the C++ interface type system and support for the C++ implementation type.

The following snippet shows a partial schema for the C++ interface element used to type services and references of component types. Additional attributes are described later in this document.

```
<interface.cpp header="NCName" class="Name"? remotable="boolean"?
            callbackHeader="NCName" callbackClass="Name"? />
```

The interface.cpp element has the following attributes:

- ***header*** – full name of the header file, including relative path from the composite root. This header file describes the interface
- ***class*** – optional name of the class declaration in the header file, including any namespace definition. If the header only contains one class then this class does not need to be specified.
- ***callbackHeader*** – optional full name of the header file that describes the callback interface, including relative path from the composite root.
- ***callbackClass*** – optional name of the class declaration for the call back in the callback header file, including any namespace definition. If the header only contains one class then this class does not need to be specified
- ***remotable*** – optional boolean value indicating whether the service is remotable or local. The default is local.

The following snippet shows a partial schema for the C++ implementation element used to define the implementation of a component. Additional attributes are described later in this document.

```
<implementation.cpp library="NCName" path="NCName"? header="NCName" class="Name"?
                    scope="scope"? />
```

The implementation.cpp element has the following attributes:

- ***library*** – name of the dll or shared library that holds the factory for the service component. This is the root name of the library. On Windows the suffix ".dll" will be appended to this name. On linux the prefix "lib" and the suffix ".so" will be added.
- ***path*** - optional path to the library which is relative to the root of the composite.
- ***header*** – The name of the header file that declares the implementation class of the component. A path is optional which is relative to the root of the composite.
- ***class*** – optional name of the class declaration of the implementation, including any namespace definition. If the header only contains one class then this class does not need to be specified.
- ***scope*** – optional attribute indicating the scope of the component implementation. The default is stateless.

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

```
// 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://www.osoa.org/xmlns/sca/1.0">
      <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://www.osoa.org/xmlns/sca/1.0"
      name="LoanComposite" >
      …

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

### 1.2.6. Instantiation

A C++ implementation class must provide a default constructor that can be used by a runtime to instantiate the component.

## *1.3. 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.

### 1.3.1. 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()*** method.

```
namespace osoa {
  namespace sca {

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

The getService() method 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() method is called on the ComponentContext passing the reference name as input. The return of the getService() method is cast to the abstract base class defined for the reference.

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

using namespace osoa::sca;

void clientMethod()
{

      unsigned long customerNumber = 1234;

      ComponentContextPtr context = ComponentContext::getCurrent();

      CustomerService* service =
               (CustomerService* )context->getService("customerService");

      short rating = service->getCreditRating(customerNumber);

}
```

### 1.3.2. 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.

### 1.3.3. Calling Service Methods

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

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

## 1.4. Error Handling

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

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

- SCAException – 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 method expecting identification of a single service is called where there are multiple services defined. Thrown by ComponentContext::getService(), ComponentContext::getSelfReference() and ComponentContext::getServiceReference().
        - ***ConversationEndedException*** – signals that a method has been called on a conversational service after the conversation was ended.
        - ***NoRegisteredCallbackException*** – signals that a callback was invoked on a service, but a callback method was not registered.

## 1.5. Conversational Services

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

Conversations occur between a client and a target service. Consequently, requests originating from one client to multiple target services will result in multiple conversations. For example, if a client A calls B and C, implemented by conversational scoped classes, two conversations will result, one between A and B and another between A and C. Likewise, requests flowing through multiple implementation instances will result in multiple conversations. For example, a request originating from A to B to C will involve two conversations (A and B, B and C). In the previous example, if a request was then made from C to A, a third conversation would result (and the implementation instance for A would be different than the one making the original request).

Callback invocations will take place within the context of the conversation.

The mechanics for maintaining and flowing conversational ids remotely is delegated to the particular binding the request is made through.

For C++ component implementations, a service interface method may be decorated with the @EndConversation annotation to indicate to the SCA runtime that the current conversation should be ended when the method is invoked. A conversation may also be ended by calling the endConversation () method on a service reference.

The following is an example interface that has been annotated as being conversational:

```
// LoanService interface
// @Remotable
// @Scope(“conversation”)
class LoanService {
public:
      virtual void apply(LoanApplication application) = 0;
      virtual void lockCurrentRate(unsigned int termInYears) = 0;
      // @EndConversation
      virtual void cancelApplication( ) = 0;
      virtual int  getLoanStatus( ) = 0;
};
```

The cancelApplication() method is annotated to end the conversation.

### 1.5.1. Conversational Client

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

```
#include "LoanApplicationClientImpl.h"
#include "ComponentContext.h"
#include "LoanService.h"
#include "LoanApplication.h"

using namespace osoa::sca;

void LoanApplicationClientImpl::clientMethod( LoanApplication loanApp,
                                              unsigned int term )
{
      unsigned long customerNumber = 1234;

      ComponentContextPtr context = ComponentContext::getCurrent();

      // service is defined as member field: LoanService* service;
      service = (LoanService* )context->getService("loanService");

      service->apply( loanApp );
      service->lockCurrentRate( term );

}

bool LoanApplicationClientImpl::isApproved()
{
      return (service->getLoanStatus() == 1);
}
```

### 1.5.2. Conversational Service Provider

The provider of the conversational service also is not required to write special code to maintain state if the implementation is annotated as conversation scoped.

There are a few capabilities that are available to the implementation of the service, but are not required. The conversation ID can be retrieved from the ServiceReference:

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

The type of the conversation ID is a std::string. Application generated conversation IDs may be other complex types, as described in the section below titled "Application Specified Conversation IDs", that are serialized to a string.

The service implementation class may also have an optional @Conversation annotation that has the following form:

```
@Conversation (maxIdleTime="10 minutes",
        maxAge="1 day",
        singlePrincipal=false)
```

The attributes of the @Conversation annotation have the following meaning:

- ***maxIdleTime*** - The maximum time that can pass between operations within a single conversation. If more time than this passes, then the container may end the conversation.
- ***maxAge*** - The maximum time that the entire conversation can remain active. If more time than this passes, then the container may end the conversation.
- ***singlePrincipal*** – If true, only the principal (the user) that started the conversation has authority to continue the operation.

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

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

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

Here is an example implementation header file of a conversational service (the implementation file is not shown).

```
// @Conversation(maxAge="30 days")
class LoanServiceImpl : public LoanService
{
public:
      virtual void apply(LoanApplication application);
      virtual void lockCurrentRate(unsigned int termInYears);
      virtual void cancelApplication(void);
      virtual int getLoanStatus();
};
```

The Conversation attributes may also be specified in the <implemention.cpp> element of a component definition:

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

### 1.5.3. Methods that End the Conversation

A method of a conversational service interface may be marked with an @EndConversation annotation. This means that once this method has been called, no further methods may be called so both the client and the target may free up resources that were associated with the conversation. It is also possible to mark a method on a callback interface (described later) as @EndConversation, in order for the service provider to be the party that chooses to end the conversation. If a method is called after the conversation completes a ConversationEndedException (which extends ServiceRuntimeException) is thrown. This may also occur if there is a race condition between the client and the service provider calling their respective @EndConversation methods.

Alternatively the **endConversation="true"** attribute can be specified on a method in the interface.cpp element of a service.

### 1.5.4. Passing Conversational Services as Parameters

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

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

```
ComponentContext::getSelfReference()
```

or

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

The second variant, which takes a *serviceName* parameter, must be used if the component implements multiple services.

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

### 1.5.5. Conversation Lifetime Summary

**Starting conversations**

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

- A service is located using ComponentContext::getService() or ComponentContext::getServices().
- A service reference is obtained using ComponentContext::getServiceReference() or ComponentContext::getServiceReferences().

**Continuing conversations**

The client can continue an existing conversation, by:

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

**Ending conversations**

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

- A service operation that has been annotated @EndConveration has been called.
- The service calls an @EndConversation method on a callback interface.
- The service's conversation lifetime timeout occurs.
- The client calls `ServiceReference::endConversation()`.
- The client calls ServiceReference::setConversationID() which implicitly ends any active conversation.

If a method is invoked on a service reference after an @EndConversation method has been called then a new conversation will automatically be started. If `ServiceReference::getConversationID()` is called after the @EndConversation method is called, but before the next conversation has been started, it will return null.

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

### 1.5.6. Application Specified Conversation IDs

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

```
    ComponentContextPtr ctx = ComponentContext::getCurrent();

    std:string conversationID("myID");

    ServiceReferencePtr serviceReference = ctx->getServiceReference("loanService");
    serviceReference->setConversationID(conversationID);

    LoanService* service = (LoanService*)serviceReference->getService();
```

The conversation ID that is passed into this method should be an instance of a std::string. The ID must be unique to the client component over all time. If the client is not an SCA component, then the ID must be globally unique.

### 1.5.7. Accessing Conversation IDs from Clients

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

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

## *1.6. 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, must be 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 method calls, callbacks, and conversational services. Each of these topics is discussed in the following sections.

### 1.6.1. 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 method that returns "void" and has no declared exceptions may be marked by using the ***@OneWay*** annotation on the method in the interface header or by using the **oneWay="true"** attribute in the interface definition of the service. This means that the method is non-blocking and communication with the service provider may use a binding that buffers the requests and sends it at some later time.

The following snippet shows an interface header for a service with the reportEvent() method declared as a one-way method:

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

      // @OneWay
      virtual void reportEvent(int eventId) = 0;

};
```

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

```
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0">
      <service name="LoanService">
            <interface.cpp header="LoanService.h">
              <method 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 recommended that service designers define one-way methods as often as possible, in order to give the greatest degree of binding flexibility to deployers.

### 1.6.2. Callbacks

A callback service is a service that is used for asynchronous communication from a service provider back to its client in contrast to the communication through return values from synchronous operations. Callbacks are used by ***bidirectional services***, which are services that have two interfaces:

1. an interface for the provided service
2. an interface that must be provided by the client

Callbacks may be used for both remotable and local services. Either both interfaces of a bidirectional service must be remotable, or both must be local. It is illegal to mix the two. There are two basic forms of callbacks: stateless callbacks and stateful callbacks.

A callback interface is declared by the **callbackHeader** and optionally **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**,

```
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0" >
      <service name="MyService">
         <interface.cpp header="MyService.h"
                     callbackHeader="MyServiceCallback.h"/>
      </service>
</componentType>
```

Alternatively the callback can be specified in the interface header using the ***@Callback*** annotation:

MyService.h file:

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

MyServiceCallback.h file:

```
// MyServiceCallback interface
class MyServiceCallback {
public:
      virtual void receiveResult( unsigned int result ) = 0;
}
```

#### 1.6.2.1. Stateful Callbacks

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

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

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

```
#include "MyServiceImpl.h"
#include "MyServiceCallback.h"
```

```
#include "osoa/sca/ComponentContext.h"
using namespace osoa::sca;

MyServiceImpl::someMethod( unsigned int arg )
{
      …
      // do some processing…

      ComponentContextPtr context = ComponentContext::getCurrent();
      ServiceReferencePtr serviceRef = context->getSelfReference();
      MyServiceCallback* callback = (MyServiceCallback* ) serviceRef-
>getCallback();
      callback->receiveResult(result);
}
```

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

```
#include "MyServiceImpl.h"
#include "MyServiceCallback.h"
#include "osoa/sca/ComponentContext.h"
using namespace osoa::sca;

void clientMethod( unsigned int arg )
{
   // locate the service
   ComponentContextPtr context = ComponentContext::getCurrent();
   MyService* service = (MyService*)context->getService("myservice");
   service->someMethod(arg);
}

MyServiceCallback::receiveResult(unsigned int result)
{
   // code to process result
}
```

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

#### *1.6.2.2. Stateless Callbacks*

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

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

```
void clientMethod( unsigned int arg )
{
      // locate the service
      ComponentContextPtr context = ComponentContext::getCurrent();
      ServiceReferencePtr svcRef = context->getServiceReference(”myservice”);
      svcRef->setCallbackID(“1234”);
      MyService* service = (MyService*)svcRef->getService();
      service->someMethod(arg);
}


MyServiceCallback::receiveResult(unsigned int result)
{
      ComponentContextPtr context = ComponentContext::getCurrent();

      ServiceReferencePtr serviceRef = context->getSelfReference();
      std::string id = serviceRef->getCallbackID();

      // code to process result
}
```

#### 1.6.2.3. 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) method must be used, passing in the name of the service for which the callback is to be obtained.

#### 1.6.2.4. Customizing the Callback

By default, the client component of a service is assumed to be the callback service for the bidirectional service. However, it is possible to change the callback by using the `ServiceReference::setCallback()` method. The object passed as the callback should implement the interface defined for the callback, including any additional SCA semantics on that interface such as its scope and whether or not it is remotable.

Since a service other than the client can be used as the callback implementation, SCA does not generate a deployment-time error if a client does not implement the callback interface of one of its references. However, if a call is made on such a reference without the setCallback() method having been called, then a NoRegisteredCallbackException will be thrown on the client.

A callback object for a stateful callback interface has the additional requirement that it must be serializable. The SCA runtime may serialize a callback object and persistently store it.

A callback object may be a service reference to another service. In that case, the callback messages go directly to the service that has been set as the callback. If the callback object is not a service reference, then callback messages go to the client and are then routed to the specific instance that has been registered as the callback object. However, if the callback interface has a stateless scope, then the callback object **must** be a service reference.

##### *1.6.2.5. Customizing the Callback Identity*

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

The callback identity has the same restrictions as the conversation ID. It must be a std::string.

## *1.7. C++ API*

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

### 1.7.1. 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.

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

```
template <typename T>
class RefCountingPointer {
public:
   T& operator* () const;
   T* operator-> () const;
```

```
    operator void* () const;
    bool operator! () const;
};
```

The RefCountingPointer class has the following methods:

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

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).

### 1.7.2. Component Context

The following shows the ComponentContext interface.

```
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 methods:

- ***getCurrent()*** – returns the ComponentContext for the current component
- ***getURI()*** – returns the absolute URI of the component.
- ***getService()*** – returns a pointer to object implementing the interface defined for the named reference. Input to the method is the name of a reference defined on this component. A MultipleServicesException will be thrown if the reference resolves to more than one service.
- ***getServices()*** – returns a list of objects implementing the interface defined for the named reference. Input to the method is the name of a reference defined on this component.
- ***getServiceReference()*** – returns a ServiceReference for the specified service. A MultipleServicesException will be thrown if the reference resolves to more than one service.
- ***getServiceReferences()*** – returns a list of ServiceReferences for the named reference.
- ***getProperties()*** – Returns an SDO from which all the properties defined in the componentType file can be retrieved.
- ***getDataFactory()*** – Returns an SDO DataFactory which can be used to create data objects.
- ***getSelfReference()*** – Returns a ServiceReference that can be used to invoke this component over the designated service. The second variant, which takes a serviceName parameter, must be used if the component implements multiple services. A MultipleServicesException is thrown if the first variant is called for a component implementing multiple services.

### 1.7.3. ServiceReference

The following shows the ServiceReference interface.

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

   virtual std::string getConversationID() const = 0;
   virtual void  setConversationID(const std::string& id) = 0;

   virtual std::string getCallbackID() const = 0;
   virtual void  setCallbackID(const std::string& id) = 0;

   virtual void* getCallback() const = 0;
   virtual void  setCallback(void* callback) = 0;

   virtual void  endConversation() = 0;

```

```
};
```

The ServiceReference interface has the following methods:

- ***getService()*** – returns a pointer to the service for this reference. A MultipleServicesException will be thrown if the reference resolves to more than one service.
- ***getConversationID()*** – returns the conversation ID
- ***setConversationID()*** – sets a user provided conversation ID
- ***getCallbackID()*** – returns the callback ID
- ***setCallbackID()*** – sets the callback ID
- ***getCallback()*** – returns the callback service
- ***setCallback()*** – sets the callback service
- ***endConversation () –*** ends the conversation for this service reference

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

### 1.7.4. SCAException

The following shows the SCAException interface.

```
namespace osoa {
     namespace sca {

          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 methods:

- ***getEClassName()*** – Returns the type of the exception as a string. e.g. “ServiceUnavailableException”.
- ***getMessageText()*** – Returns the message which the SCA runtime attached to the exception.
- ***getFileName()*** – Returns the filename within which the exception occurred – May be zero if the filename is not known.
- ***getLineNumber()*** – Returns the line number at which  the exception occurred.

- ***getFunctionName()*** – Returns the function name in which the exception occurred.

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

### 1.7.5. SCANullPointerException

The following shows the SCANullPointerException interface.

```
namespace osoa {
      namespace sca {

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

### 1.7.6. ServiceRuntimeException

The following shows the ServiceRuntimeException interface.

```
namespace osoa {
      namespace sca {

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

### 1.7.7. ServiceUnavailableException

The following shows the ServiceUnavailableException interface.

```
namespace osoa {
      namespace sca {

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

### 1.7.8. NoRegisteredCallbackException

The following shows the NoRegisteredCallbackException interface.

```
namespace osoa {
      namespace sca {

        class NoRegisteredCallbackException : public ServiceRuntimeException {
            };
      }
}
```

### 1.7.9. ConversationEndedException

The following shows the ConversationEndedException interface.

```
namespace osoa {
      namespace sca {

        class ConversationEndedException : public ServiceRuntimeException {
            };
      }
}
```

### 1.7.10. MultipleServicesException

The following shows the MultipleServicesException interface.

```
namespace osoa {
      namespace sca {

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

## 1.8. C++ Annotations

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

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

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

### 1.8.1. Interface Header Annotations

This section lists the annotations that may be used in the header file that defines a service interface. These annotations can also be represented in SCDL within the <interface.cpp> element.

#### 1.8.1.1. @Remotable

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

Format:

```
// @Remotable
```

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

Interface header:

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

Service definition:

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

#### 1.8.1.2. @Callback

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

Format:

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

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

Interface header:

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

```
};
```

Service definition:

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

#### 1.8.1.3. @OneWay

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

Format:

```
// @OneWay
```

Interface header:

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

Service definition:

```
<service name="LoanService">
      <interface.cpp header="LoanService.h">
           <method name=”reportEvent” oneWay=”true” />
      </interface.cpp>
</service>
```

#### 1.8.1.4. @EndConversation

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

Format:

```
// @EndConversation
```

Interface header:

```
class LoanService
{
public:
      // @EndConversation
      virtual void cancelApplication( ) = 0;
      …
};
```

Service definition:

```
<service name="LoanService">
      <interface.cpp header="LoanService.h">
           <method name=" cancelApplication" endConversation="true" />
      </interface.cpp>
</service>
```

### 1.8.2. Implementation Header Annotations

This section lists the annotations that may be used in the header file that defines a service implementation. These annotations can also be represented in SCDL within the <implementation.cpp> element.

#### *1.8.2.1. @Scope*

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

Format:

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

where **value** can be **stateless**, **composite**, **request** or **conversation**. The default value is **stateless**.

Implementation header:

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

Component definition:

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

#### 1.8.2.2. @EagerInit

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

Format:

```
// @EagerInit
```

Implementation header:

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

Component definition:

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

#### 1.8.2.3. @AllowsPassByReference

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

Format:

```
// @AllowsPassByReference
```

Implementation header:

```
// @AllowsPassByReference
class LoanService {
        ...
};
```

Component definition:

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

#### 1.8.2.4. @Conversation

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

Format:

// @Conversation(maxIdleTime=”**value**”, maxAge=”**value**”, singlePrincipal=**boolValue**)

where **value** is a time expressed as an integer followed by a space and then one of the following: "seconds", "minutes", "hours", "days" or "years".

Implementation header:

```
// @Conversation(maxAge="30 days", maxIdleTime=”5 minutes”,
//                singlePrincipal=false)
class LoanServiceImpl : public LoanService
{
        …
};
```

Component definition:

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

#### 1.8.2.5. @Property

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

Format:

// @Property(name=“propertyName”, type=”typeQName”

//              default=”defaultValue”, required**=**”true”)

where

- **name (optional)** 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 (optional)** 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 the appendix **C++ to WSDL Mapping**. If the data member is an array, then the property is many-valued.
- **required (optional)** specifies whether a value must be set in the component definition for this property. Default is **false**
- **default (optional)** specifies a default value and is only needed if **required** is **false**,

Implementation:

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

Component Type definition:

```
<componentType … >
      <service … />
      <property name="loanType" type="xsd:int" />
</componentType>
```

### 1.8.2.6. @Reference

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

Format:

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

where

- **name (optional)** 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 (required)** specifies the C++ header defining the interface for the reference.
- **interfaceClass (optional)** 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 (optional)** specifies whether a value must 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.

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>
```

## 1.9. WSDL to C++ and C++ to WSDL Mapping

The SCA Client and Implementation Model for C++ applies the ***WSDL to C++*** mapping rules as defined by the OMG **WSDL to C++ Mapping Specification** [3] and the C++ to WSDL mapping rules as defined in the appendix **C++ to WSDL Mapping**.

# 2. Appendix 1

## 2.1. Packaging

### 2.1.1. Composite Packaging

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

```
MyValue/
      MyValue.composite
```

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

```
MyValue/
      MyValue.composite

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

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

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

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

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

# 3. Appendix 2

## 3.1. 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.

### 3.1.1. 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)

### 3.1.2. 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.

Not supported:

- Pointers.
- References.

# 4. Appendix 3

## 4.1. Restrictions on C++ header files

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

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

The following C++ keywords and constructs must not be used:

- Macros
- inline methods
- friend classes

# 5. Appendix 4

## 5.1. XML Schemas

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

### 5.1.1. sca-interface-cpp.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://www.commonj.org/xmlns/sca/1.0/"
         xmlns:sca="http://www.commonj.org/xmlns/sca/1.0/"
         xmlns:sdo="commonj.sdo/XML"
         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=”method” type=”sca:CPPMethod”
                                 minOccurs=0 maxOccurs=”unbounded” />
                           <any namespace="##other" processContents="lax"
                                        minOccurs="0" maxOccurs="unbounded"/>
                     </sequence>
                     <attribute name="header" type="NCName" use="required"/>
                     <attribute name="class" type="Name" use="required"/>
                     <attribute name="callbackHeader" type="NCName"
                           use="optional"/>
                     <attribute name="callbackClass" type="Name" use="optional"/>
                     <attribute name="remotable" type="boolean" use="optional"/>
                     <anyAttribute namespace="##any" processContents="lax"/>
               </extension>
         </complexContent>
   </complexType>

   <complexType name="CPPMethod">
         <complexContent>
               <attribute name="name" type="NCName" use="required"/>
               <attribute name="oneWay" type="boolean" use="optional"/>
               <attribute name="endConversation" type="boolean" use="optional"/>
               <anyAttribute namespace="##any" processContents="lax"/>
         </complexContent>
   </complexType>

</schema>
```

### 5.1.2. sca-implementation-cpp.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema  xmlns="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
         xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
         xmlns:sdo="commonj.sdo/XML"
         elementFormDefault="qualified">

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

   <element name="implementation.cpp" type="sca:CPPImplementation"
          substitutionGroup="sca:implementation” />
   <complexType name="CPPImplementation">
      <complexContent>
            <extension base="sca:Implementation">
                  <sequence>
                        <element name=”method”
                              type=”sca:CPPImplementationMethod”
                              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="NCName" use="optional"/>
                  <attribute name="class" type="Name" use="optional"/>
                  <attribute name="scope" type="sca:CPPImplementationScope"
                     use="optional"/>
                  <attribute name="eagerInit" type="boolean" use="optional"/>
                  <attribute name="allowsPassByReference" type="boolean"
                                                            use="optional"/>
                  <attribute name="conversationMaxAge" type="string"
                     use="optional"/>
                  <attribute name="conversationMaxIdle" type="string"
                     use="optional"/>
                  <attribute name="conversationSinglePrincipal" type="boolean"
                     use="optional"/>
                  <anyAttribute namespace="##any" processContents="lax"/>
            </extension>
      </complexContent>
   </complexType>

   <simpleType name="CPPImplementationScope">
      <restriction base="string">
            <enumeration value="stateless"/>
            <enumeration value="composite"/>
            <enumeration value="request"/>
            <enumeration value="converstion"/>
      </restriction>
   </simpleType>

   <complexType name="CPPImplementationMethod">
      <complexContent>
            <attribute name="name" type="NCName" use="required"/>
            <attribute name="allowsPassByReference" type="boolean"
                                                      use="optional"/>
```

```
                <anyAttribute namespace="##any" processContents="lax"/>
            </complexContent>
        </complexType>

    </schema>
```

# 6. Appendix 5

## 6.1. C++ to WSDL Mapping

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

This mapping currently only deals with producing document/literal wrapped style services and WSDL from C++ header files.

## 6.2. Parameter and Return Type mappings

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

### 6.2.1. Built-in, STL and SDO type mappings

| *C++ built in, STL and SDO types* | *Notes* | *XML Type* |
|---|---|---|
| bool | | xsd:boolean |
| char | signed 8-bit[1] | xsd:byte |
| unsigned char | unsigned 8-bit[1] | xsd:unsignedByte |
| short | signed 16-bit[1] | xsd:short |
| unsigned short | unsigned 16-bit[1] | xsd:unsignedShort |

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

| | | |
|---|---|---|
| int | signed 16-bit[1] | xsd:short |
| unsigned int | unsigned 16-bit[1] | xsd:unsignedShort |
| long | signed 32-bit[1] | xsd:int |
| unsigned long | unsigned 32-bit[1] | xsd:unsignedInt |
| long long | signed 64-bit[1] | xsd:long |
| unsigned long long | unsigned 64-bit[1] | xsd:unsignedLong |
| float | 32-bit floating point (IEEE-754-1985)[1] | xsd:float |
| double | 64-bit floating point (IEEE-754-1985)[1] | xsd:double |
| long double | 64-bit floating point (platform dependent, IEEE-754-1985)[1] | xsd:double |
| char* or char array | Must be a null-terminated UTF- | xsd:string |

| | | |
|---|---|---|
| | 8 encoded string | |
| wchar_t* or wchar_t array[2] | Must be a null-terminated UTF-16 or UTF-32 encoded string | xsd:string |
| std::string | Must be a UTF-8 encoded string | xsd:string |
| std::wstring[2] | Must be a UTF-16 or UTF-32 encoded string | xsd:string |
| commonj::sdo::DataObjectPtr | | xsd:any |

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

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

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

would generate a schema like:

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

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

### 6.2.2. Binary data mapping

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

### 6.2.3. Array mapping

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

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

would generate a schema like:

```
<xsd:element name="myMethod">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="idList" type="xsd:int"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="value" type="xsd:double"/>
```

```
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
```

### 6.2.4. Multi-dimensional array mapping

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

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

would generate a schema like:

```
<xsd:element name="myMethod">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="multiIdArray"
                   minOccurs="0" maxOccurs="unbounded"/>
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="multiIdArray"
                         minOccurs="4" maxOccurs="4"/>
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element name="multiIdArray" type="xsd:int"
                               minOccurs="2" maxOccurs="2" />
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

### 6.2.5. Pointer/reference mapping

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

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

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

would generate a schema like:

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

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

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

### 6.2.6. STL container mapping

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

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

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

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

could convert to a schema like:

```
<xsd:element name="myMethod">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="myMap" type="DataMap"
             minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="id" type="xsd:int"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

```
<xsd:complexType name="DataMap">
  <xsd:sequence>
    <xsd:element name="data1" type="xsd:string"/>
    <xsd:element name="data2" type="xsd:double"/>
  </xsd:sequence>
</xsd:complexType>
```

### 6.2.7. Struct mapping

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

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

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

```
struct DataStruct {
  std::string name;
  double value;
};
```

could convert to a schema like:

```
<xsd:element name="myMethod">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="data" type="DataStruct" />
      <xsd:element name="id" type="xsd:int"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

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

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

### 6.2.8. Enum mapping

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

For example, a method such as:

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

with the ParameterType type defined as an enum:

```
enum ParameterType
{
   UNSET = 1,
   TYPEA,
   TYPEB,
   TYPEC
};
```

could convert to a schema like:

```
<xsd:element name="getValueFromType">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="type" type="ParameterType"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:simpleType name="ParameterType">
  <xsd:restriction base="xsd:int">
    <xs:minInclusive value="1"/>
    <xs:maxInclusive value="4"/>
  </xsd:restriction>
</xsd:simpleType>
```

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

```
enum ParameterType
{
   UNSET = 'u',
   TYPEA = 'A',
```

```
    TYPEB = 'B',
    TYPEC = 'C'
};
```

could convert to a schema like:

```
<xsd:simpleType name="ParameterType">
  <xsd:restriction base="xsd:int">
    <xsd:enumeration value="86"/> <!-- Character 'u' -->
    <xsd:enumeration value="65"/> <!-- Character 'A' -->
    <xsd:enumeration value="66"/> <!-- Character 'B' -->
    <xsd:enumeration value="67"/> <!-- Character 'C' -->
  </xsd:restriction>
</xsd:simpleType>
```

### 6.2.9. Union mapping

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

### 6.2.10. Typedef mapping

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

### 6.2.11. Pre-processor mapping

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

### 6.2.12. Nesting types

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

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

with types defined as follows:

```
struct DataStruct {
  std::string name;
  ValuesVector values;
  ParameterType type;
};
```

```
typedef std::vector<double> ValuesVector;

enum ParameterType
{
   UNSET = 1,
   TYPEA,
   TYPEB,
   TYPEC
};
```

would convert to a schema like:

```
<xsd:element name="myMethod">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="data" type="DataStruct"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:complexType name="DataStruct">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="values" type="ValuesVector"/>
    <xsd:element name="type" type=" ParameterType"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ValuesVector">
  <xsd:sequence>
    <xsd:element name="data" type="xsd:double"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:simpleType name="ParameterType">
  <xsd:restriction base="xsd:int">
    <xs:minInclusive value="1"/>
    <xs:maxInclusive value="4"/>
  </xsd:restriction>
</xsd:simpleType>
```

#### 6.2.13. SDO mapping

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

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

would generate a schema like:

```
<xsd:element name="myMethod">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="data">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:any processContents="skip"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
     </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

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

#### 6.2.14. void* mapping

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

#### 6.2.15. User-defined types (UDT) mapping

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

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

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

```
class AnObject
{
  public:
    AnObject();

```

```
        std::string getMyString() const;
        double getMyDouble() const;

        void setMyString(std::string data);
        void setMyDouble(double otherData);
    };
```

would generate a schema like:

```
<xsd:element name="myMethod">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="data" type="AnObject"/>
     </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:complexType name="AnObject">
  <xsd:sequence>
    <xsd:element name="MyString" type="xsd:string"/>
    <xsd:element name="MyDouble" type="xsd:double"/>
  </xsd:sequence>
</xsd:complexType>
```

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

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

### 6.2.16. Included or Inherited types

All types (structs, enums, classes, etc) that need to be mapped to WSDL schema must be able to be found from the C++ header being mapped. Implementations should allow a list of “include” directories to be specified. Types that are included (via a #include “SomeHeader.h” statement) or inherited from a superclass must be mappable to a schema element via the rules in this document.

## 6.3. Namespace mapping

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

```
namespace myCorp
{
    namespace myServices
```

```
    {
        class ExampleService
        {
        public:
           // Methods go here
        };
    }
}
```

would generate WSDL like:

```
<definitions name="ExampleService"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    targetNamespace="http://myCorp/myServices/ExampleService"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema”>
  <types>
    <xsd:schema
        targetNamespace="http://myCorp/myServices/ExampleService"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
...
```

Implementations should allow namespace mappings to be specified separately to override this default behaviour.

## 6.4. Class mapping

A single class in a C++ header maps to a single WSDL service element, a single WSDL binding element and a single WSDL portType element. The WSDL service element contains a single WSDL port element. The WSDL binding and WSDL portType elements each contain multiple WSDL operation elements that map to the public methods defined in the C++ class. A pair of WSDL message elements and a pair of XML schema elements are generated for each WSDL operation. SOAP 1.1 binding and address information is also generated. For example, a C++ header such as:

```
class MyService
{
  public:
    int myMethod(std::string data);
    double myOtherMethod(double otherData);
};
```

would generate WSDL like:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://MyService"
  targetNamespace="http://MyService">
  <types>
    <xsd:schema targetNamespace="http://sample/MyService"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:tns="http://MyService" elementFormDefault="qualified">

      <xsd:element name="myMethod">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="data" type="xsd:string"/>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="myMethodResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="myMethodResponseData" type="xsd:int"/>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="myOtherMethod">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="otherData" type="xsd:double"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="myOtherMethodResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="myMethodResponseData" type="xsd:double"/>
        </xsd:complexType>
      </xsd:element>

    </xsd:schema>
  </types>

```

```
    <message name="myMethodRequestMsg">
      <part name="body" element="tns:myMethod"/>
    </message>
    <message name="myMethodResponseMsg">
      <part name="body" element="tns:myMethodResponse"/>
    </message>

    <message name="myOtherMethodRequestMsg">
      <part name="body" element="tns:myOtherMethod"/>
    </message>
    <message name="myOtherMethodResponseMsg">
      <part name="body" element="tns:myOtherMethodResponse"/>
    </message>

    <portType name="MyServicePortType">
      <operation name="myMethod">
        <input message="tns:myMethodRequestMsg"/>
        <output message="tns:myMethodResponseMsg"/>
      </operation>
      <operation name="myOtherMethod">
        <input message="tns:myOtherMethodRequestMsg"/>
        <output message="tns:myOtherMethodResponseMsg"/>
      </operation>
    </portType>

    <binding name="MyServiceBinding" type="tns:MyService">
      <soap:binding style="document"
          transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="myMethod">
        <soap:operation soapAction="MyService#myMethod"/>
        <input>
          <soap:body use="literal"/>
        </input>
        <output>
          <soap:body use="literal"/>
        </output>
      </operation>
      <operation name="myOtherMethod">
        <soap:operation soapAction="MyService#myOtherMethod"/>
        <input>
          <soap:body use="literal"/>
        </input>
```

```
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>

  <service name="MyService">
    <port name="MyServicePort" binding="tns:MyServiceBinding">
      <soap:address location="http://server:9090/MyService"/>
    </port>
  </service>
</definitions>
```

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

This specification requires support for generating a SOAP 1.1, document/literal style binding. Support for additional bindings (such as SOAP 1.2) is not required, however if provided should be consistent with the SOAP 1.1 binding specified in this document. Support for additional binding styles is implementation dependent.

## 6.5. Method mapping

### 6.5.1. Default parameter value mapping

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

E.g. a method prototype:

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

would generate a schema like:

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

### 6.5.2. Non-named parameters and the return type

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

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

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

### 6.5.3. The void return type

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

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

would generate a schema like:

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

<xsd:element name="myMethodResponseMsg">
  <xsd:complexType/>
</xsd:element>
```

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

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

```
      <input message="tns:myMethodRequestMsg"/>
      <output message="tns:myMethodResponseMsg"/>
    </operation>
  </portType>

  <binding name="MyServiceBinding" type="tns:MyService">
    <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="myMethod">
      <soap:operation soapAction="MyService#myMethod"/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
```

Alternatively, if the oneWay annotation is specified on the method:

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

the following schema would be generated:

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

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

```
<portType name="MyServicePortType">
  <operation name="myMethod">
    <input message="tns:myMethodRequestMsg"/>
```

```
    </operation>
  </portType>

  <binding name="MyServiceBinding" type="tns:MyService">
    <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="myMethod">
      <soap:operation soapAction="MyService#myMethod"/>
      <input>
        <soap:body use="literal"/>
      </input>
    </operation>
  </binding>
```

### 6.5.4. No Parameters Specified

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

```
int getValue();
```

would generate a schema like:

```
<xsd:element name="getValue">
  <xsd:complexType/>
</xsd:element>

<xsd:element name="getValueResponse">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="getValueResponseData" type="xsd:int"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

### 6.5.5. In/Out Parameters

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

### 6.5.6. Public Methods

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

### 6.5.7. Inherited Public Methods

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

### 6.5.8. Protected / Private Methods

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

### 6.5.9. Constructors / Destructors

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

### 6.5.10. Overloaded Methods

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

### 6.5.11. Operator overloading

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

### 6.5.12. Exceptions

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

# 7. References

[1] SDO 2.1 Specification
http://www.osoa.org/display/Main/Service+Data+Objects+Specifications

[2] SCA Assembly Specification
http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications

[3] OMG WSDL to C++ Mapping Specification
http://www.omg.org/docs/ptc/06-08-01.pdf