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

Committee Draft 01

20 March 2008

Specification URIs:

This Version:

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

Previous Version:

N/A

Latest Version:

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

Latest Approved Version:

Technical Committee:

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

Chair:

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

Editors:

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

Related work:

This specification replaces or supercedes:

- OSOA SCA C++ Client and Implementation V1.00

This specification is related to:

- OASIS Service Component Architecture Assembly Model Version 1.1
- OASIS SCA Policy Framework Version 1.1
- OASIS Service Component Architecture Web Service Binding Specification Version 1.1

Declared XML Namespace(s):

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

Copyright © OASIS® 2005, 2008. All Rights Reserved.

Abstract:

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

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

Status:

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

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

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

Copyright © OASIS® 2005, 2008. All Rights Reserved.

Notices

Copyright © OASIS® 2005, 2008. All Rights Reserved.

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

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

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

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

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

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

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

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

Table of Contents

1 Introduction 7
1.1 Terminology 7
1.2 Normative References 7
1.3 Non-Normative References 7
2 Basic Component Implementation Model 8
2.1 Implementing a Service 8
2.1.1 Implementing a Remotable Service 9
2.1.2 Implementing a Local Service 10
2.2 Conversational and Non-Conversational services 10
2.3 Component Implementation Scopes 10
2.3.1 Stateless scope 11
2.3.2 Request scope 11
2.3.3 Composite scope 11
2.3.4 Conversation scope 11
2.4 Implementing a Configuration Property 12
2.5 Component Type and Component 12
2.5.1 Interface.cpp 13
2.5.2 Method 14
2.5.3 Implementation.cpp 14
2.5.4 Implementation Method 15
2.6 Instantiation 16
3 Basic Client Model 17
3.1 Accessing Services from Component Implementations 17
3.2 Accessing Services from non-SCA component implementations 18
3.3 Calling Service Methods 18
4 Error Handling 19
5 Conversational Services 20
5.1 Conversational Client 20
5.2 Conversational Service Provider 20
5.3 Methods that End the Conversation 21
5.4 Passing Conversational Services as Parameters 22
5.5 Conversation Lifetime Summary 22
5.6 Application Specified Conversation IDs 23
5.7 Accessing Conversation IDs from Clients 23
6 Asynchronous Programming 24
6.1 Non-blocking Calls 24
6.2 Callbacks 24
6.2.1 Stateful Callbacks 25
6.2.2 Stateless Callbacks 26
6.2.3 Implementing Multiple Bidirectional Interfaces 26
6.2.4 Customizing the Callback Identity 26
7 C++ API 27
7.1 Reference Counting Pointers 27

Copyright © OASIS® 2005, 2008. All Rights Reserved.

7.2 Component Context 27
7.3 ServiceReference....28
7.4 SCAException 29
7.5 SCANullPointerException 29
7.6 ServiceRuntimeException 30
7.7 ServiceUnavailableException 30
7.8 NoRegisteredCallbackException 30
7.9 ConversationEndedException....30
7.10 MultipleServicesException 31
8 WSDL to C++ and C++ to WSDL Mapping 32
9 Packaging 33
9.1 Composite Packaging 33
10 Types Supported in Service Interfaces....34
10.1 Local service 34
10.2 Remotable service 34
11 Restrictions on C++ header files....35
12 C++ to WSDL Mapping....36
12.1 Parameter and Return Type mappings....36
12.1.1 Built-in, STL and SDO type mappings 36
12.1.2 Binary data mapping 37
12.1.3 Array mapping 37
12.1.4 Multi-dimensional array mapping 38
12.1.5 Pointer/reference mapping 38
12.1.6 STL container mapping 39
12.1.7 Struct mapping 39
12.1.8 Enum mapping 40
12.1.9 Union mapping 41
12.1.10 Typedef mapping....41
12.1.11 Pre-processor mapping 41
12.1.12 Nesting types....41
12.1.13 SDO mapping 42
12.1.14 void* mapping....43
12.1.15 User-defined types (UDT) mapping 43
12.1.16 Included or Inherited types 44
12.2 Namespace mapping 44
12.3 Class mapping 44
12.4 Method mapping 46
12.4.1 Default parameter value mapping 46
12.4.2 Non-named parameters and the return type 47
12.4.3 The void return type 47
12.4.4 No Parameters Specified 49
12.4.5 In/Out Parameters 49
12.4.6 Public Methods 49
12.4.7 Inherited Public Methods....49
12.4.8 Protected/Private Methods 49

Copyright © OASIS® 2005, 2008. All Rights Reserved.

12.4.9 Constructors/Destructors 49
12.4.10 Overloaded Methods 49
12.4.11 Operator overloading 49
12.4.12 Exceptions 49
13 Conformance 50
A C++ Annotations 51
A.1 Application of Annotations to C++ Program Elements 51
A.2 Interface Header Annotations 51
A.2.1 @Interface 51
A.2.2 @Remotable 52
A.2.3 @Callback 52
A.2.4 @OneWay 53
A.2.5 @Conversational 54
A.2.6 @EndsConversation 54
A.3 Implementation Header Annotations 55
A.3.1 @ComponentType 55
A.3.2 @Scope 55
A.3.3 @EagerInit 56
A.3.4 @AllowsPassByReference 56
A.3.5 @ConversationAttributes 57
A.3.6 @Property 58
A.3.7 @Reference 58
B Policy Annotations for C++ 60
B.1 General Intent Annotations 60
B.2 Specific Intent Annotations 61
B.3 Application of Intent Annotations 62
B.4 Inheritance and Intent Annotations 62
B.5 Relationship of Declarative and Annotated Intents 64
B.6 Policy Set Annotations 64
B.7 Security Policy Annotations 65
B.7.1 Security Interaction Policy 65
B.7.2 Security Implementation Policy 66
C XML Schemas 69
C.1 sca-interface-cpp-1.1-schema.xsd 69
C.2 sca-implementation-cpp-1.1-schema.xsd 69
D Migration 71
D.1 Annotations related to conversations 71
E Acknowledgements 72
F Non-Normative Text 73
G Revision History 74

Copyright © OASIS® 2005, 2008. All Rights Reserved.

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 Terminology

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

1.2 Normative References

[RFC2119] S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, http://www.ietf.org/rfc/rfc2119.txt, IETF RFC 2119, March 1997.

[ASSEMBLY] M. Beisiegel, et al., *Service Component Architecture Assembly Model Specification Version 1.1*, http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec.pdf, OASIS Service Component Architecture Assembly Model Specification Version 1.1, XXX 2008

[POLICY] J. Anderson, et al., *SCA Policy Framework Version 1.1*, http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec.pdf, OASIS SCA Policy Framework Version 1.1, XXX 2008

[SDO21] J. Beatty, et al., *Service Data Objects For C++ Specification*, http://www.osoa.org/download/attachments/36/CPP-SDO-Spec-v2.1.0-FINAL.pdf, SDO 2.1, December 2006.

[WSDL11] E. Christensen, et al., Web Service *Description Language (WSDL)*, http://www.w3.org/TR/wsdl, W3C Note Web Service Description Language (WSDL), March 2001

[WSDL20] R. Chinnici, et al., *Web Service Description Language (WSDL) Vesrsion 2.0 Part 1: Core Language*, http://www.w3.org/TR/wsdl20/, W3C Web Service Description Language (WSDL) Version 2.0 Part 1: Core Language, June 2007

1.3 Non-Normative References

[CPPWSDL] *WSDL to C++ Mapping Specification*, http://www.omg.org/docs/ptc/06-08-01.pdf, OMG WSDL to C++ Mapping Specification, August 2006

Copyright © OASIS® 2005, 2008. All Rights Reserved.

2 Basic Component Implementation Model

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

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

2.1 Implementing a Service

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

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

- a C++ abstract base class
- a WSDL 1.1 portType **[WSDL11]**
- a WSDL 2.0 interface **[WSDL20]**

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 all of the operations of the service interface.

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.

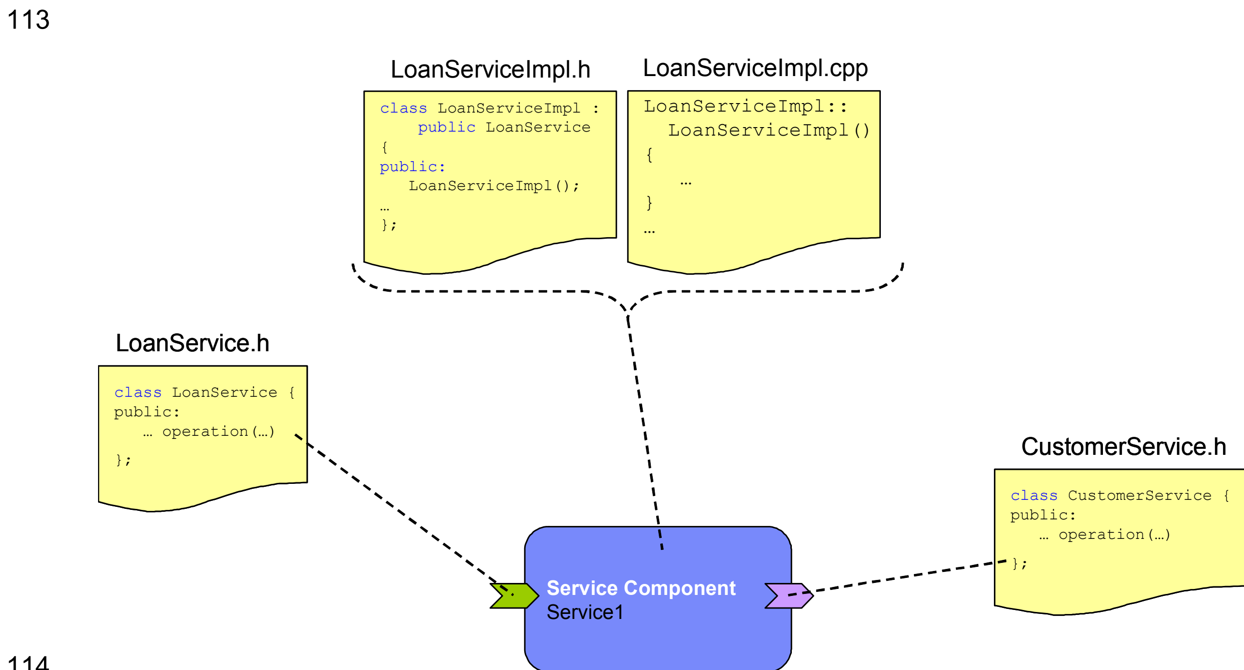
Copyright © OASIS® 2005, 2008. All Rights Reserved.

```
#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://docs.oasis-open.org/ns/opencsa/sca/200712">
   <service name="LoanService">
          <interface.cpp header="LoanService.h"/>
   </service>
</componentType>
```

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



2.1.1 Implementing a Remotable Service

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

Copyright © OASIS® 2005, 2008. All Rights Reserved.

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

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

An implementation of a remotable service can declare whether it allows pass by reference data exchange semantics on calls to it, meaning that the by-value semantics can be maintained without requiring that the parameters be copied. 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=true* attribute on the implementation.cpp element of a remotable service is used to declare that calls to the whole interface allows pass by reference. Alternatively, this attribute can be used on a specific method.

2.1.2 Implementing a Local Service

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

2.2 Conversational and Non-Conversational services

A *requires="conversational"* attribute on an interface.cpp element indicates that the service contract is conversational, as described in the Assembly Specification **[ASSEMBLY]**.

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.

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

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

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>
```


Copyright © OASIS® 2005, 2008. All Rights Reserved.

For *stateless* scoped implementations, the SCA runtime MUST 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 scopes an SCA runtime MUST support for C++ component implementations.

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.

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.

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=”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.

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 Methods that End the Conversation). A conversation is potentially long-running and the SCA runtime MAY choose to passivate implementation instances. If this occurs, the runtime MUST guarantee implementation instance state is preserved.

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

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.

Copyright © OASIS® 2005, 2008. All Rights Reserved.

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. 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”);

   …
}
```

2.5 Component Type and Component

For a C++ component implementation, a component type is specified in a side file. By default, the componentType side file is in the same composite directory as the header file for the implementation class with a name matching this header file. The location and name can be modified as described below.

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

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

```
// 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);
};
```


Copyright © OASIS® 2005, 2008. All Rights Reserved.

Implementation.

```
#include "LoanServiceImpl.h"

////////////////////////////////////////////////////////////////////////
// Construction/Destruction
////////////////////////////////////////////////////////////////////////
LoanServiceImpl::LoanServiceImpl()
{
   …
}
LoanServiceImpl::~LoanServiceImpl()
{
   …
}
////////////////////////////////////////////////////////////////////////
// Implementation
////////////////////////////////////////////////////////////////////////
bool LoanServiceImpl::approveLoan(unsigned long customerNumber,
 unsigned long loanAmount)
{
   …
}
```

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

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

The following snippet shows the component using the implementation.

```
<?xml version="1.0" encoding="ASCII"?>
<composite      xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
   name="LoanComposite" >
   …

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

2.5.1 Interface.cpp

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

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

   <method … />*

```


Copyright © OASIS® 2005, 2008. All Rights Reserved.

```
</interface.cpp>
```

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

- ***header : NCName (1..1) –*** full name of the header file, including relative path from the composite root. This header file describes the interface
- ***class : Name (0..1) –*** 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 : NCName (0..1) –*** full name of the header file that describes the callback interface, including relative path from the composite root.
- ***callbackClass : Name (0..1) –*** name of the class declaration for the 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 : boolean (0..1) –*** indicates whether the service is remotable or local. The default is local.

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

method : CPPMethod (0..n) – see Method

2.5.2 Method

Some methods of an interface have behavioral characteristics, which will be described later, that need to be identified. This is done using a method child element of interface.cpp

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

```
<?xml version="1.0" encoding="ASCII"?>
<!—- Method schema snippet -->
<interface.cpp xmlns=http://docs.oasis-open.org/ns/opencsa/sca/200712 … >

   <method name=”NCName” oneWay=”Boolean”? endsConversation=”Boolean”? />*

</interface.cpp>
```

The ***method*** element has the following ***attributes***:

- ***name : NCName (1..1) –*** name of the method being decorated.
- ***oneWay : boolean (0..1) –*** see Non-blocking Calls
- ***endsConversation : boolean (0..1) –*** see Methods that End the Conversation

2.5.3 Implementation.cpp

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

```
<?xml version="1.0" encoding="ASCII"?>
<!—- implementation.cpp schema snippet -->
<implementation.cpp xmlns=“http://docs.oasis-open.org/ns/opencsa/sca/200712“
      library="NCName" path=”Name”? header="NCName" class=”Name”?
      scope="scope"? componentType=”NCName”? allowsPassByReference=Boolean”?
      eagerInit=”boolean”? conversationMaxAge=”string”?
      conversationMaxIdle=”string”? conversationSinglePrincipal=”Boolean”? >

   <method … />*

```


Copyright © OASIS® 2005, 2008. All Rights Reserved.

```
</implementation.cpp>
```

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

- ***library : NCName (1..1) –*** name of the dll or shared library that holds the factory for the service component. This is the root name of the library. On Windows the suffix ".dll" will be appended to this name. On linux the prefix "lib" and the suffix ".so" will be added.
- ***path : Name (0..1) -*** path to the library which is either absolute or relative to the root of the composite.
- ***header : NCName (1..1) –*** 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 : Name (0..1) –*** 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, and the header file name SHOULD be the same as the class name.
- ***scope : CPPImplementationScope (0..1) –*** indentifies the scope of the component implementation. The default is stateless.
- ***componentType : NCName (0..1) –*** path to the componentType file which is relative to the root of the composite.
- ***allowsPassByReference : boolean (0..1) –*** indicates the service allows pass by reference data exchange semantics on calls to it.
- ***eagerInit type : boolean (0..1) –*** indicates a composite scoped implementation should be initialized when it is loaded.
- ***conversationMaxAge : string (0..1) –*** see Conversational Service Provider**.**
- ***conversationMaxIdle : string (0..1) –*** see Conversational Service Provider**.**
- ***conversationSinglePrincipal : boolean (0..1) –*** see Conversational Service Provider**.**

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

method : CPPImplementationMethod (0..n) – see Implementation Method

If the class attribute is specified on implementation.cpp, then the SCA runtime MUST use class name as the name of the componentType file. The SCA runtime will append ".componentType" to the class name to find the componentType file.

2.5.4 Implementation Method

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

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

```
<?xml version="1.0" encoding="ASCII"?>
<!-- ImplementationMethod schema snippet -->
<implementation.cpp xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" … >

   <method name="NCName" allowsPassByReference="boolean"? />*

</implementation.cpp>
```

The ***method*** element has the following ***attributes***:

- ***name : NCName (1..1) –*** name of the method being decorated.

Copyright © OASIS® 2005, 2008. All Rights Reserved.

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

412 2.6 Instantiation

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

Copyright © OASIS® 2005, 2008. All Rights Reserved.

3 Basic Client Model

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

3.1 Accessing Services from Component Implementations

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

The following snippet shows the ComponentContext C++ class with its *getService()* 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);

}
```


Copyright © OASIS® 2005, 2008. All Rights Reserved.

3.2 Accessing Services from non-SCA component implementations

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

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

3.3 Calling Service 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 it is possible to get a *ServiceUnavailableException*, which is a ServiceRuntimeException.

Copyright © OASIS® 2005, 2008. All Rights Reserved.

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.

Copyright © OASIS® 2005, 2008. All Rights Reserved.

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 might be the series of service calls that are necessary in order to apply for a bank loan.

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);
}
```

5.2 Conversational Service Provider

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

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


Copyright © OASIS® 2005, 2008. All Rights Reserved.

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

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

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 definition of a conversational service.

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

5.3 Methods that End the Conversation

A method of a conversational service interface can be marked with an *endsConversation=”true”* attribute. This means that once this method has been called, no further methods are to be called, allowing both the client and the target to free up resources that were associated with the conversation. It is also possible to mark a method on a callback interface (described later) as endsConversation=”true”, 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 can also occur if there is a race condition between the client and the service provider calling their respective endsConversation methods. Calling the endConversation() method on a service reference also ends a conversation.

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

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

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

Copyright © OASIS® 2005, 2008. All Rights Reserved.

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, is used if the component implements multiple services.

This can 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.

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 marked endsConveration=”true” is called.
- The service calls a method marked endsConversation=”true” 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 a method marked endsConversation=”true” has been called then a new conversation will automatically be started. If `ServiceReference::getConversationID()` is called after a method marked endsConversation=”true” is called, but before the next conversation has been started, it will return null.

Copyright © OASIS® 2005, 2008. All Rights Reserved.

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 has to be called.

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 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.

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

5.7 Accessing Conversation IDs from Clients

Whether the conversation ID is chosen by the client or is generated by the system, the client accesses 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.

Copyright © OASIS® 2005, 2008. All Rights Reserved.

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, is fed back to the client through a separate mechanism, since no output is available at the point where the service is invoked. This is in contrast to the call-and-return style of synchronous programming, where the invoked service executes and returns any output to the client before the client continues. The SCA asynchronous programming model consists of support for non-blocking method calls, callbacks, and conversational services. Each of these topics is discussed in the following sections.

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 can be marked by using the *oneWay="true"* attribute in the interface definition of the service. This means that the method is non-blocking and the SCA runtime MAY use a binding that buffers the requests and sends it at some later time.

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

```
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
   <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.

6.2 Callbacks

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

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

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


Copyright © OASIS® 2005, 2008. All Rights Reserved.

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 normally is *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 to be used.

Copyright © OASIS® 2005, 2008. All Rights Reserved.

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
}
```

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.

6.2.4 Customizing the Callback Identity

The identity that is used to identify a callback request is, by default, generated by the SCA runtime. However, it is possible to provide an application specified identity to 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 SCA runtime will send the ID back when any callback method is invoked.

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

Copyright © OASIS® 2005, 2008. All Rights Reserved.

7 C++ API

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

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).

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;
```


Copyright © OASIS® 2005, 2008. All Rights Reserved.

```
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 **[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, is used if the component implements multiple services. A MultipleServicesException is thrown if the first variant is called for a component implementing multiple services.

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  endConversation() = 0;

};
```


Copyright © OASIS® 2005, 2008. All Rights Reserved.

926
927 The ServiceReference interface has the following methods:
928
929 • *getService()* – returns a pointer to the service for this reference. A MultipleServicesException will be
930 thrown if the reference resolves to more than one service.
931 • *getConversationID()* – returns the conversation ID
932 • *setConversationID()* – sets a user provided conversation ID
933 • *getCallbackID()* – returns the callback ID
934 • *setCallbackID()* – sets the callback ID
935 • *getCallback()* – returns the callback service
936 • *endConversation ()* – ends the conversation for this service reference
937
938 The detailed description of the usage of these methods is described in the sections on Conversational
939 Services and Asynchronous Programming in this document.

940 7.4 SCAException

941 The following shows the SCAException interface.
942

```
943 namespace osoa {
944   namespace sca {
945
946          class SCAException : public std::exception {
947          public:
948                 const char* getEClassName() const;
949                 const char* getMessageText() const;
950                 const char* getFileName() const;
951                 unsigned long getLineNumber() const;
952                 const char* getFunctionName() const;
953          };
954   }
955 }
```

956
957 The SCAException C++ interface has the following methods:
958 • *getEClassName()* – Returns the type of the exception as a string. e.g. “ServiceUnavailableException”.
959 • *getMessageText()* – Returns the message which the SCA runtime attached to the exception.
960 • *getFileName()* – Returns the filename within which the exception occurred – Will be zero if the
961 filename is not known.
962 • *getLineNumber()* – Returns the line number at which the exception occurred.
963 • *getFunctionName()* – Returns the function name in which the exception occurred.
964
965 Details concerning this class and its derived types are described in the section on Error Handling in this
966 document.

967 7.5 SCANullPointerException

968 The following shows the SCANullPointerException interface.
969

```
970 namespace osoa {
```


Copyright © OASIS® 2005, 2008. All Rights Reserved.

971 972 973 974 975 976

```
   namespace sca {

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

977

7.6 ServiceRuntimeException

979 The following shows the ServiceRuntimeException interface.

980

981 982 983 984 985 986 987

```
namespace osoa {
   namespace sca {

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

7.7 ServiceUnavailableException

989 The following shows the ServiceUnavailableException interface.

990

991 992 993 994 995 996 997

```
namespace osoa {
   namespace sca {

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

7.8 NoRegisteredCallbackException

999 The following shows the NoRegisteredCallbackException interface.

1000

1001 1002 1003 1004 1005 1006 1007 1008

```
namespace osoa {
   namespace sca {

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

1009

7.9 ConversationEndedException

1011 The following shows the ConversationEndedException interface.

1012

1013 1014 1015 1016 1017

```
namespace osoa {
   namespace sca {

            class ConversationEndedException : public ServiceRuntimeException {
          };
```


Copyright © OASIS® 2005, 2008. All Rights Reserved.

```
   }
}
```

7.10 MultipleServicesException

The following shows the MultipleServicesException interface.

```
namespace osoa {
   namespace sca {

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


Copyright © OASIS® 2005, 2008. All Rights Reserved.

1030 # 8 WSDL to C++ and C++ to WSDL Mapping

1031 The SCA Client and Implementation Model for C++ applies the *WSDL to C++* mapping rules as defined
1032 by the OMG *WSDL to C++ Mapping Specification* **[CPPWSDL]** and the C++ to WSDL mapping rules as
1033 defined in C++ to WSDL Mapping.

Copyright © OASIS® 2005, 2008. All Rights Reserved.

9 Packaging

9.1 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.

Copyright © OASIS® 2005, 2008. All Rights Reserved.

10 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.

10.1 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)

10.2 Remotable service

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

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

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

- Pointers.
- References.

Copyright © OASIS® 2005, 2008. All Rights Reserved.

1106 # 11 Restrictions on C++ header files

1107 A C++ header file that is used to describe an interface has some restrictions. It MUST:

1108
- Declare at least one class with:

1109
  - At least one public method.

1110
  - All public methods must be pure virtual (virtual with no implementation)

1111

1112 The following C++ keywords and constructs MUST NOT be used:

1113
- Macros

1114
- inline methods

1115
- friend classes

Copyright © OASIS® 2005, 2008. All Rights Reserved.

1116 # 12 C++ to WSDL Mapping

1117 This section describes a mapping from C++ header interfaces to a WSDL description of that interface.
1118 The intent is for implementations of this proposal to be able to deploy a service based only on a C++
1119 header definition and for a WSDL definition of that service to be generated from the C++, either at deploy
1120 or run time.

1121

1122 This mapping currently only deals with producing document/literal wrapped style services and WSDL from
1123 C++ header files. Support for additional styles,if provided SHOULD be consistent with the mapping
1124 specified in this specification. Support for additional styles is implementation dependent.

1125 ## 12.1 Parameter and Return Type mappings

1126 This section details how types used as parameters or return types in C++ method prototypes get mapped
1127 to XML schema elements in the generated WSDL.

1128 ### 12.1.1 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
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	A null-terminated UTF-8 encoded string	xsd:string
wchar_t* or wchar_t array[2]	A null-terminated UTF-16 or UTF-32	xsd:string

[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.

Copyright © OASIS® 2005, 2008. All Rights Reserved.

	encoded string	
std::string	A UTF-8 encoded string	xsd:string
std::wstring[2]	A UTF-16 or UTF-32 encoded string	xsd:string
Commonj::sdo::DataObjectPtr		xsd:any

1129

1130 For example, a C++ method prototype defined in a header such as:

1131

```
1132 long myMethod(char* name, int id, double value);
```

1133

1134 would generate a schema like:

1135

```
1136 <xsd:element name="myMethod">
1137   <xsd:complexType>
1138     <xsd:sequence>
1139       <xsd:element name="name" type="xsd:string"/>
1140       <xsd:element name="id" type="xsd:short"/>
1141       <xsd:element name="value" type="xsd:double"/>
1142     </xsd:sequence>
1143   </xsd:complexType>
1144 </xsd:element>
1145
1146 <xsd:element name="myMethodResponse">
1147   <xsd:complexType>
1148     <xsd:sequence>
1149       <xsd:element name="myMethodResponseData" type="xsd:int"/>
1150     </xsd:sequence>
1151   </xsd:complexType>
1152 </xsd:element>
```

1153 12.1.2 Binary data mapping

1154 Binary data, such as data passed via non-null-terminated char* or char arrays, is not supported in this
1155 mapping. char* and char array parameters and return types are always mapped to xsd:string, and are
1156 null-terminated. This requirement also applies to wchar_t* and wchar_t array parameters.

1157 12.1.3 Array mapping

1158 C++ arrays passed in or out of methods get mapped as normal elements but with multiplicity allowed via
1159 the minOccurs and maxOccurs attributes. E.g. a method prototype such as

1160

```
1161 long myMethod(char* name, int idList[], double value);
```

1162

1163 would generate a schema like:

1164

```
1165 <xsd:element name="myMethod">
1166   <xsd:complexType>
1167     <xsd:sequence>
1168       <xsd:element name="name" type="xsd:string"/>
```

[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.

Copyright © OASIS® 2005, 2008. All Rights Reserved.

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

12.1.4 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:short"
                               minOccurs="2" maxOccurs="2" />
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

12.1.5 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:short"/>
      <xsd:element name="value" type="xsd:double"/>
```


Copyright © OASIS® 2005, 2008. All Rights Reserved.

```
    </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.

12.1.6 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:short"/>
    </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>
```

12.1.7 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;
};
```


Copyright © OASIS® 2005, 2008. All Rights Reserved.

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:short"/>
    </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.

12.1.8 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>
```


Copyright © OASIS® 2005, 2008. All Rights Reserved.

```
</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>
```

12.1.9 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.

12.1.10 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.

12.1.11 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.

12.1.12 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;
```


Copyright © OASIS® 2005, 2008. All Rights Reserved.

```
  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>
```

12.1.13 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>
```


Copyright © OASIS® 2005, 2008. All Rights Reserved.

```
          <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.

12.1.14 void* mapping

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

12.1.15 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>
```


Copyright © OASIS® 2005, 2008. All Rights Reserved.

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.

12.1.16 Included or Inherited types

An implementation MUST completely map a C++ interface to WSDL, including types (structs, enums, classes, etc) that might be defined in a different C++ header than the one that is 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.

12.2 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 behavior.

12.3 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 binding and address information is also generated. For example, a C++ header such as:

Copyright © OASIS® 2005, 2008. All Rights Reserved.

```
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://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:short"/>
        </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="myOtherMethodResponseData" 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"/>
```


Copyright © OASIS® 2005, 2008. All Rights Reserved.

```
  </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.

12.4 Method mapping

12.4.1 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:

Copyright © OASIS® 2005, 2008. All Rights Reserved.

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

12.4.2 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, an implementation MUST generate a name 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:short"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

12.4.3 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">
```


Copyright © OASIS® 2005, 2008. All Rights Reserved.

```
    <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 attribute is specified on the method:

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

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>
```


Copyright © OASIS® 2005, 2008. All Rights Reserved.

12.4.4 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:short"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

12.4.5 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.

12.4.6 Public Methods

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

12.4.7 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.

12.4.8 Protected/Private Methods

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

12.4.9 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.

12.4.10 Overloaded Methods

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

12.4.11 Operator overloading

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

12.4.12 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.

Copyright © OASIS® 2005, 2008. All Rights Reserved.

1796

13 Conformance

Copyright © OASIS® 2005, 2008. All Rights Reserved.

A C++ Annotations

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

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

```
// @Scope(“stateless”)
```

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

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

- Class

  The annotation immediately precedes the class.

  Example

```
// @Scope(“composite”)
class LoanServiceImpl : public LoanService {
   …
};
```

- Method

  The annotation immediately precedes the method.

  Example:

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

- Data Member

  The annotation immediately precedes the data member.

  Example:

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

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

A.2 Interface Header Annotations

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

A.2.1 @Interface

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

Copyright © OASIS® 2005, 2008. All Rights Reserved.

1839

1840 **Corresponds to:** class attribute of interface.cpp element.

1841

1842 **Format:**

```
1843 // @Interface
```

1844

1845 **Applies to:** Class

1846

1847 Example

1848 Interface header:

```
1849 // @Interface
1850 class LoanService {
1851 ...
1852 };
```

1853

1854 Service definition:

```
1855 <service name="LoanService">
1856   <interface.cpp header="LoanService.h" class="LoanService" />
1857 </service>
```

1858 A.2.2 @Remotable

1859 Annotation on service interface class to indicate that a service is remotable.

1860

1861 **Corresponds to:** remotable="true" attribute of interface.cpp element.

1862

1863 **Format:**

```
1864 // @Remotable
```

1865 The default is *false* (not remotable).

1866

1867 **Applies to:** Class

1868

1869 Example

1870 Interface header:

```
1871 // @Remotable
1872 class LoanService {
1873 ...
1874 };
```

1875

1876 Service definition:

```
1877 <service name="LoanService">
1878   <interface.cpp header="LoanService.h" remotable="true" />
1879 </service>
```

1880 A.2.3 @Callback

1881 Annotation on a service interface class to specify the callback interface.

1882

1883 **Corresponds to:** callbackHeader and callbackClass attributes of interface.cpp element.

Copyright © OASIS® 2005, 2008. All Rights Reserved.

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.

Applies to: Class

Example

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>
```

A.2.4 @OneWay

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

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

Format:

```
// @OneWay
```

Applies to: Method

Example

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>
```


Copyright © OASIS® 2005, 2008. All Rights Reserved.

A.2.5 @Conversational

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

Corresponds to: requires=”conversational” attribute of a service element.

Format:

```
// @Conversational
```

Applies to: Class

Example

Interface header:

```
//@Conversational
class LoanService
{
   …
}
```

Service definition:

```
<service name=”LoanService” requires=”conversational”>
   <interface.cpp header=”LoanService.h”>
</service>
```

A.2.6 @EndsConversation

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

Corresponds to: endsConversationy=”true” attribute of method element of an interface.cpp element.

Format:

```
// @EndsConversation
```

Applies to: Method

Example

Interface header:

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

Service definition:

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


Copyright © OASIS® 2005, 2008. All Rights Reserved.

```
   </interface.cpp>
</service>
```

A.3 Implementation Header Annotations

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

A.3.1 @ComponentType

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

Corresponds to: class attribute of implementation.cpp element.

Format:

```
// @ComponentType
```

Applies to: Class

Example

Implementation header:

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

Component definition:

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

A.3.2 @Scope

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

Corresponds to: scope attribute of implementation.cpp element.

Format:

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

where

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

Applies to: Class

Example

Implementation header:

Copyright © OASIS® 2005, 2008. All Rights Reserved.

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

Component definition:

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

A.3.3 @EagerInit

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

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

Format:

```
// @EagerInit
```

Applies to: Class

Example

Implementation header:

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

Component definition:

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

A.3.4 @AllowsPassByReference

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

Corresponds to: allowsPassByReference="true" attribute of implementation.cpp element or a method element of an implementation.cpp element.

Format:

```
// @AllowsPassByReference
```

Applies to: Class or Method

Example

Copyright © OASIS® 2005, 2008. All Rights Reserved.

Implementation header:

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

Component definition:

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

A.3.5 @ConversationAttributes

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

Corresponds to: conversationMaxAge, conversationMaxIdle or conversationSinglePrincipal attributes of implementation.cpp element.

Format:

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

where

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

Applies to: Class

Example

Implementation header:

```
// @ConversationAttributes(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>
```


Copyright © OASIS® 2005, 2008. All Rights Reserved.

2115 ## A.3.6 @Property

2116 Annotation on a service implementation class data member to define a property of the service.

2117

2118 **Corresponds to:** property element of componentType element.

2119

2120 **Format:**

```
2121 // @Property(name="propertyName", type="typeQName"
2122 //                  default="defaultValue", required="true")
```

2123 where

2124 • ***name : NCName (0..1) -*** specifies the name of the property. If name is not specified the property
2125 name is taken from the name of the following data member.

2126 • ***type : QName (0..1) -*** specifies the type of the property. If not specified the type of the property is
2127 based on the C++ mapping of the type of the following data member to an xsd type as defined in C++
2128 to WSDL Mapping. If the data member is an array, then the property is many-valued.

2129 • ***required : boolean (0..1) -*** specifies whether a value has to be set in the component definition for this
2130 property. Default is *false*

2131 • ***default : <type> (0..1) -*** specifies a default value and is only needed if *required* is *false*,

2132

2133 **Applies to**: DataMember

2134

2135 Example

2136 Implementation:

```
2137 // @Property(name="loanType", type="xsd:int")
2138 long loanType;
```

2139

2140 Component Type definition:

```
2141 <componentType … >
2142    <service … />
2143    <property name="loanType" type="xsd:int" />
2144 </componentType>
```

2145 ## A.3.7 @Reference

2146 Annotation on a service implementation class data member to define a reference of the service.

2147

2148 **Corresponds to:** reference element of componentType element.

2149

2150 **Format:**

```
2151 // @Reference(name="referenceName", interfaceHeader="LoanService.h",
2152 //                     interfaceClass="LoanService", required="true")
```

2153 where

2154 • ***name : NCName (0..1) -*** specifies the name of the reference. If name is not specified the reference
2155 name is taken from the name of the following data member.

2156 • ***interfaceHeader : Name (1..1) -*** specifies the C++ header defining the interface for the reference.

2157 • ***interfaceClass : Name (0..1) -*** specifies the C++ class defining the interface for the reference. If not
2158 specified the class is derived from the type of the annotated data member.

2159 • ***required : boolean (0..1) -*** specifies whether a value has to be set for this reference. Default is *true.*

Copyright © OASIS® 2005, 2008. All Rights Reserved.

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

Applies to: Data Member

Example

Implementation:

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

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

Component Type definition:

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


Copyright © OASIS® 2005, 2008. All Rights Reserved.

B Policy Annotations for C++

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

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

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

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

B.1 General Intent Annotations

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

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

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

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

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

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

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


Copyright © OASIS® 2005, 2008. All Rights Reserved.

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

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

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

Format:

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

where

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

Applies to: Class, Method, Data Member

Examples

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

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

A reference requiring support for confidentiality:

```
class Foo {
   …
// @Requires(CONFIDENTIALITY)
// @Reference(interfaceHeader="SetBar.h")
   void setBar(Bar* bar);
   …
}
```

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

```
class Foo {
   …
// @Requires(SCA_PREFIX "confidentiality")
// @Reference(interfaceHeader="SetBar.h")
   void setBar(Bar* bar);
   …
}
```

B.2 Specific Intent Annotations

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

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

Copyright © OASIS® 2005, 2008. All Rights Reserved.

2278 For example, the SCA confidentiality intent described in General Intent Annotations using the
2279 @Requires(CONFIDENTIALITY) intent can also be specified with the specific @Confidentiality intent
2280 annotation. The specific intent annotation for the "integrity" security intent is:

```
2281 // @Integrity
```

2282

2283 **Corresponds to:** requires="<Intent>" attribute of a service, reference, operation or property element.

2284

2285 **Format:**

```
2286 // @<Intent>[(qualifiers)]
```

2287 where Intent is an NCName that denotes a particular type of intent.

```
2288 Intent ::= NCName
2289 qualifiers ::= "qualifier" | {"qualifier" [, "qualifier"] }
2290 qualifier ::= NCName | NCName/qualifier
```

2291

2292 **Applies to:** Class, Method, Data Member – but see specific intents for restrictions

2293

2294 Example

```
2295 // @Authentication( {"message", "transport"} )
```

2296 This annotation attaches the pair of qualified intents: "authentication.message" and
2297 "authentication.transport" (the sca: namespace is assumed in both of these cases – "http://docs.oasis-
2298 open.org/ns/opencsa/sca/200712").

2299 B.3 Application of Intent Annotations

2300 Where multiple intent annotations (general or specific) are applied to the same C++ element, they are
2301 additive in effect. An example of multiple policy annotations being used together follows:

2302

```
2303 // @Authentication
2304 // @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
```

2305

2306 In this case, the effective intents are "authentication", "confidentiality.message" and "integrity.message".

2307

2308 If an annotation is specified at both the class/interface level and the method or field level, then the method
2309 or field level annotation completely overrides the class level annotation of the same type.

2310

2311 The intent annotation can be applied either to classes or to class methods when adding annotated policy
2312 on SCA services. Applying an intent to the setter method in a reference injection approach allows intents
2313 to be defined at references.

2314 B.4 Inheritance and Intent Annotations

2315 The following example shows the inheritance relations of intents on classes, operations, and super
2316 classes.

2317

```
2318 // @Remotable
2319 // @Integrity("transport")
2320 // @Authentication
2321 class HelloService {
2322 public:
2323 // @Integrity
```


Copyright © OASIS® 2005, 2008. All Rights Reserved.

```
2324 // @Authentication(“message”)
2325    wchar_t* hello(wchar_t* message) {...}
2326
2327 // @Integrity
2328 // @Authentication(“transport”)
2329    wchar_t* helloThere() {...}
2330 }
2331
2332 // @Remotable
2333 // @Confidentiality(“message”)
2334 class HelloChildService : public HelloService {
2335 public:
2336 // @Confidentiality(“transport”)
2337    wchar_t* hello(wchar_t* message) {...}
2338 // @Authentication
2339    wchar_t* helloWorld(){...}
2340 }
```

2341 Example 1a. Usage example of annotated policy and inheritance.

2342

- 2343 The effective intent annotation on the helloWorld method is Integrity(“transport”), @Authentication,
  2344 and @Confidentiality(“message”).
- 2345 The effective intent annotation on the hello method of the HelloChildService is @Integrity(“transport”),
  2346 @Authentication, and @Confidentiality(“transport”),
- 2347 The effective intent annotation on the helloThere method of the HelloChildService is @Integrity and
  2348 @Authentication(“transport”), the same as in HelloService class.
- 2349 The effective intent annotation on the hello method of the HelloService is @Integrity and
  2350 @Authentication(“message”)

2351

2352 The listing below contains the equivalent declarative security interaction policy of the HelloService and
2353 HelloChildService implementation corresponding to the C++ classes shown in Example 1a.

2354

```
2355 <?xml version="1.0" encoding="ASCII"?>
2356
2357 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2358                 name="HelloServiceComposite" >
2359    <service name=”HelloService” requires=”integrity/transport
2360           authentication”>
2361           …
2362    </service>
2363    <service name=”HelloChildService” requires=”integrity/transport
2364           authentication confidentiality/message”>
2365    …
2366    </service>
2367    ...
2368
2369    <component name="HelloServiceComponent">*
2370           <implementation.cpp library="HelloService.dll"
2371                  header=”HellowServiceImpl.h”/>
2372           <operation name=”hello” requires=”integrity
2373                  authentication/message”/>
2374           <operation name=”helloThere” requires=”integrity
2375                  authentication/transport”/>
2376    </component>
2377    <component name="HelloChildServiceComponent">*
2378           <implementation.cpp library="HelloChildService.dll"
2379                  header=”HelloChilDServiceImpl.h” />
2380           <operation name=”hello” requires=”confidentiality/transport”/>
2381           <operation name=”helloThere” requires=” integrity/transport
```


Copyright © OASIS® 2005, 2008. All Rights Reserved.

```
                 authentication"/>
            <operation name=helloWorld" requires="authentication"/>
      </component>

      ...

</composite>
```

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

B.5 Relationship of Declarative and Annotated Intents

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

a declarative intent in a using composite document.

B.6 Policy Set Annotations

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

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

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

Format:

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

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

Applies to: Class, Method, Data Member

Example

```
// @Reference(name="helloService", interfaceHeader="helloService.h",
//              required=true)
// @PolicySets({ MY_NS "WS_Encryption_Policy",
//                 MY_NS "WS_Authentication_Policy"})
   HelloService* helloService;
   …
```

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

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

Copyright © OASIS® 2005, 2008. All Rights Reserved.

B.7 Security Policy Annotations

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

B.7.1 Security Interaction Policy

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

- @Integrity
- @Confidentiality
- @Authentication

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

- message
- transport

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

```
//Interface for HelloService
class HelloService {
public:
   virtual wchar_t* hello(wchar_t* helloMsg);
}

// Interface for ClientService
class ClientService {
public:
   virtual void clientMethod();
}

// Implementation class for ClientService
#include "ComponentContext.h"
#include "HelloService.h"

class ClientServiceImpl : public ClientService {

private:
// @Reference(name="helloService", interfaceHeader="HelloService.h")
// @Integrity("message")
// @Authentication("message")
   HelloService* helloService;


public:
   void clientMethod() {
      ComponentContextPtr context = ComponentContext::getCurrent();

      helloService = (HelloService* )context->getService("helloService");

      wchar_t* result = helloService->hello(L"Hello World!");

   }
}
```


Copyright © OASIS® 2005, 2008. All Rights Reserved.

B.7.2 Security Implementation Policy

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

- @RunAs

  Takes as a parameter a string which is the name of a Security role, e.g. @RunAs("Manager").

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

- @RolesAllowed

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

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

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

- @PermitAll

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

- @DenyAll

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

- @DeclareRoles

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

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

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

B.7.2.1 Annotated Implementation Policy Example

The following is an example showing annotated security implementation policy:

```
// @Remotable
class AccountService {
public:
   virtual AccountReport* getAccountReport(char* customerID);
}
```

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

```
#include “commonj.sdo.h”;
#include “AccountDataService.h”;
#include “CheckingAccount.h”;
#include “SavingsAccount.h”;
#include “StockAccount.h”;
#include “StockQuoteService.h”;

// @RolesAllowed(“customers”)
// @RunAs(“accountants” )
class AccountServiceImpl : public AccountService {
```


Copyright © OASIS® 2005, 2008. All Rights Reserved.

```
protected:
// @Property
   std::string currency = "USD";

// @Reference(name="accountDataService",
//             interfaceHeader=”AccountDataService.h”)
   AccountDataService accountDataService;

// @Reference(name="stockQuoteService",
//             interfaceHeader=”StockQuoteService.h”)
   StockQuoteService stockQuoteService;

public:
// @RolesAllowed({“customers”, “accountants”})
   AccountReport getAccountReport(char* customerID) {

      DataFactory dataFactory = DataFactory::getDataFactory();
      AccountReport accountReport =
                      (AccountReport)dataFactory ->create(AccountReportType);
      DataObjectList accountSummaries = accountReport->getAccountSummaries();

      CheckingAccount checkingAccount =
                          accountDataService->getCheckingAccount(customerID);
      AccountSummary checkingAccountSummary =
                    (AccountSummary)dataFactory ->create(AccountSummaryType);
      checkingAccountSummary->setAccountNumber(
                                      checkingAccount->getAccountNumber());
      checkingAccountSummary->setAccountType("checking");
      checkingAccountSummary->setBalance(fromUSDollarToCurrency(
                                       checkingAccount->getBalance()));
      accountSummaries->append(checkingAccountSummary);

      SavingsAccount savingsAccount =
                           accountDataService->getSavingsAccount(customerID);
      AccountSummary savingsAccountSummary =
                     (AccountSummary)dataFactory->create(AccountSummaryType);
      savingsAccountSummary->setAccountNumber(
                                       savingsAccount->getAccountNumber());
      savingsAccountSummary->setAccountType("savings");
      savingsAccountSummary->setBalance(fromUSDollarToCurrency(
                                      savingsAccount->getBalance()));
      accountSummaries->append(savingsAccountSummary);

      StockAccount stockAccount =
                             accountDataService->getStockAccount(customerID);
      AccountSummary stockAccountSummary =
                     (AccountSummary)dataFactory->create(AccountSummaryType);
      stockAccountSummary->setAccountNumber(stockAccount->getAccountNumber());
      stockAccountSummary->setAccountType("stock");
      float balance = (stockQuoteService->getQuote(
                                         stockAccount->getSymbol())) *
                                         stockAccount->getQuantity();
      stockAccountSummary->setBalance(fromUSDollarToCurrency(balance));
      accountSummaries->append(stockAccountSummary);

      return accountReport;
   }

// @PermitAll
   float fromUSDollarToCurrency(float value){

      if (currency == "USD"))
         return value;
      else if (currency == "EURO"))
```


Copyright © OASIS® 2005, 2008. All Rights Reserved.

```
            return value * 0.8;
        else
            return 0.0;
        }
}
```

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

- @RolesAllowed(“customers”) - indicating that customers have access to the implementation as a whole
- @RunAs(“accountants”) – indicating that the code in the implementation runs with the permissions of accountants

The getAccountReport(..) method is marked with @RolesAllowed({“customers”, “accountants”}), which indicates that this method can be called by both customers and accountants.

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

Copyright © OASIS® 2005, 2008. All Rights Reserved.

C XML Schemas

Two XML schemas are defined to support the use of C++ for implementation and definition of interfaces. The normative schemas are located at:

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

and

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

The following copies are provided for reference.

C.1 sca-interface-cpp-1.1-schema.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
      xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
      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="##other" processContents="lax"/>
         </extension>
      </complexContent>
   </complexType>

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

</schema>
```

C.2 sca-implementation-cpp-1.1-schema.xsd

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


Copyright © OASIS® 2005, 2008. All Rights Reserved.

```
2659        targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2660        xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2661        xmlns:sdo="commonj.sdo/XML"
2662        elementFormDefault="qualified">
2663
2664     <include schemaLocation="sca-core.xsd"/>
2665
2666     <element name="implementation.cpp" type="sca:CPPImplementation"
2667           substitutionGroup="sca:implementation" />
2668     <complexType name="CPPImplementation">
2669        <complexContent>
2670           <extension base="sca:Implementation">
2671              <sequence>
2672                 <element name="method" type="sca:CPPImplementationMethod"
2673                       minOccurs="0" maxOccurs="unbounded" />
2674                 <any namespace="##other" processContents="lax"
2675                       minOccurs="0" maxOccurs="unbounded"/>
2676              </sequence>
2677              <attribute name="library" type="NCName" use="required"/>
2678              <attribute name="header" type="NCName" use="required"/>
2679              <attribute name="path" type="Name" use="optional"/>
2680              <attribute name="class" type="Name" use="optional"/>
2681              <attribute name="componentType" type="NCName" use="optional"/>
2682              <attribute name="scope" type="sca:CPPImplementationScope"
2683                    use="optional"/>
2684              <attribute name="eagerInit" type="boolean" use="optional"/>
2685              <attribute name="allowsPassByReference" type="boolean"
2686                    use="optional"/>
2687              <attribute name="conversationMaxAge" type="string"
2688                    use="optional"/>
2689              <attribute name="conversationMaxIdle" type="string"
2690                    use="optional"/>
2691              <attribute name="conversationSinglePrincipal" type="boolean"
2692                    use="optional"/>
2693              <anyAttribute namespace="##other" processContents="lax"/>
2694           </extension>
2695        </complexContent>
2696     </complexType>
2697
2698     <simpleType name="CPPImplementationScope">
2699        <restriction base="string">
2700           <enumeration value="stateless"/>
2701           <enumeration value="composite"/>
2702           <enumeration value="request"/>
2703           <enumeration value="converstion"/>
2704        </restriction>
2705     </simpleType>
2706
2707     <complexType name="CPPImplementationMethod">
2708        <attribute name="name" type="NCName" use="required"/>
2709        <attribute name="allowsPassByReference" type="boolean"
2710              use="optional"/>
2711        <anyAttribute namespace="##other" processContents="lax"/>
2712     </complexType>
2713
2714  </schema>
```


Copyright © OASIS® 2005, 2008. All Rights Reserved.

D Migration

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

D.1 Annotations related to conversations

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

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

Copyright © OASIS® 2005, 2008. All Rights Reserved.

2722 # E Acknowledgements

2723 The following individuals have participated in the creation of this specification and are gratefully
2724 acknowledged:

2725 **Participants:**
2726 Andrew Borley, IBM
2727 Bryan Aupperle, IBM
2728 David Haney, Rogue Wave Software
2729 Jeff Mischkinsky, Oracle
2730 Mike Edwards, IBM
2731 Pete Robbins, IBM

Copyright © OASIS® 2005, 2008. All Rights Reserved.

2732 # F Non-Normative Text

Copyright © OASIS® 2005, 2008. All Rights Reserved.

2733 # G Revision History

2734 [optional; should not be included in OASIS Standards]

2735

Revision	Date	Editor	Changes Made
5	2008-3-17	Bryan Aupperle	• Editorial changes in preparation for Committee Draft
4	2008-2-26	Bryan Aupperle	• Remove duplicated text from Assembly Spec • Reformat pseudo-schema presentation and description to be consistent with assembly spec • Incorporate changes for CCPP-32, CCPP-34 and CCPP-35
3	2008-1-24	Bryan Aupperle	• Conformance statement cleanup • Incorporate changes for CCPP-21, CCPP-26 and CCPP-31
2	2007-11-02	David Haney, Bryan Aupperle	• Add initial abstract (based on Introduction), updated Editors list, corrected typo in Acknowledgements. • Incorporate changes proposed in CCPP-12, CCPP-13, CCPP-14. • Update MUST/must, SHOULD/should and MAY/may to reflect TC use or RFC 2119 • Incorporate changes for CCPP-1, CCPP-4, CCPP-10, CCPP-15 CCPP-18 and CCPP-19
1	2007-09-22	David Haney	Initial conversion of the OSOA 1.0 specification to the OASIS template.

2736

Copyright © OASIS® 2005, 2008. All Rights Reserved.