SCA Service Component Architecture

Client and Implementation Model Specification for C++

SCA Version 0.9, November 2005

Technical Contacts:	Mike Greenberg	IONA Technologies
	Todd Little	BEA Systems, Inc.
	Brian Lorenz	Sybase, Inc.
	Pete Robbins	IBM Corporation
	Colin Thorne	IBM Corporation
	Steve Vinoski	IONA Technologies

Copyright Notice

License

Status of this Document

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

Table of Contents

1. Client and Implementation Model

1.1. Introduction

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

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

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

1.2. Basic Component Implementation Model

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

Every component implementation is itself also a client of services, this aspect of a component implementation is described in the basic client model section.

1.2.1. Implementing a Service

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

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

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

There is the intention in a future specification to specify how a C++ base class can be generated from a WSDL portType.

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

Service interface.

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

Implementation declaration header file.

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

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

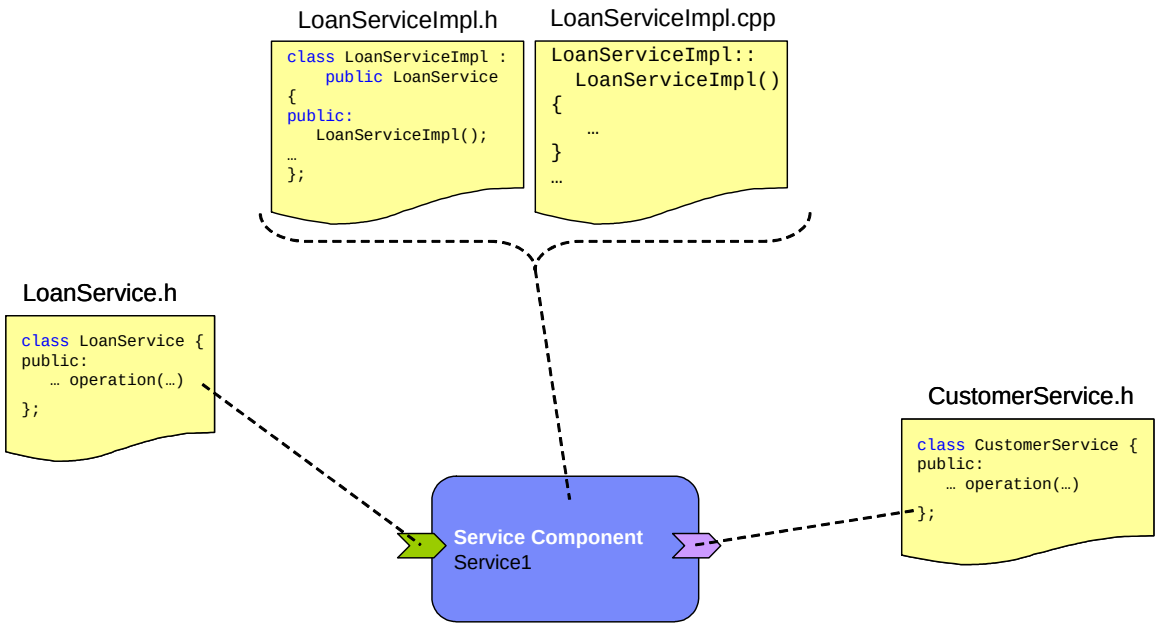
Implementation.

```
#include "LoanServiceImpl.h"


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

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

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

```
<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/0.9">
   <service name="LoanService">
         <interface.cpp header="LoanService.h"/>
   </service>
</componentType>
```

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



1.2.1.1. *Implementing a Remotable Service*

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

Whether a service is remotable is defined by the interface of the service. In the case of C++ this is defined by adding the ***remotable*** attribute to the C++ interface definition in the componentType. The following snippet shows the component type for a remotable service:

```
<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/0.9">
   <service name="LoanService">
         <interface.cpp header="LoanService.h" remotable="true"/>
   </service>
</componentType>
```

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

Complex data types exchanged via remotable service interfaces must be compatible with the marshalling technology that is used by any binding that is used for the service. For example, if the service is going to be exposed using the standard web service binding, then the parameters must be Service Data Objects (SDOs) 2.0 [1].

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

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

1.2.1.2. Implementing a Local Service

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

Whether a service is local is defined by the interface of the service. In the case of C++ this is defined by the remotable attribute being set to false or not being present on the interface definition in the component type file.

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

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

1.2.1.3. Implementing the Stateful Resource Pattern

The fact that a service has its state maintained by SCA is represented by the scope attribute on the interface definition in the component type file. SCA refers to such a service as a scoped service.

Clients accessing a scoped service will get a service instance according to scope. The following are the currently supported scope values:

- ***stateless*** *(default)* – Each request is handled separately. C++ class instances may be drawn from a pool of instances that are use to service requests.

- ***module*** – Service requests are delegated to the same C++ class instance for all requests within the same "module". The lifecycle of a module-scoped instance extends from the point where the module component is loaded in the SCA runtime such that it is ready to service requests until the module is unloaded or stopped. Registry lookups will return the same module-scoped service instance for the duration of the module.

Note: The scope model is extensible and allows for new scopes to be defined. A future version of the specifications will describe how to extend an SCA runtime to handle additional scopes.

The following snippet shows the component type for a module scoped component.

```
<componentType xmlns="http://www.osoa.org/xmlns/sca/0.9">
   <service name="LoanService">
         <interface.cpp header="LoanService.h" scope="module"/>
   </service>
</componentType>
```

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

1.2.2. 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->getLong("maxLoanValue");

   …

}
```

1.2.3. Component Type and Component

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

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

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

```
<interface.cpp header="Name" class=”Name”? remotable="boolean"? scope="scope"/>
```

The interface.cpp element has the following attributes:

- ***header*** – full name of the header file, including relative path from the module root. This header file describes the interface
- ***class*** – optional name of the class declaration in the header file, including any namespace definition. If the header only contains one class then this class does not need to be specified.
- ***remotable*** – optional boolean value indicating whether the service is remotable or local. The default is local.
- ***scope*** – optional attribute indicating the scope of the component implementation. The default is stateless.

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

```
<implementation.cpp dll="NCName" header="NCName" class=”Name”? />
```

The implementation.cpp element has the following attributes:

- ***dll*** – name of the dll or shared library that holds the factory for the service component. The name of the dll may specify a path which is relative to the root of the module.
- ***header*** – The name of the header file that declares the implementation class of the component. A path is optional which is relative to the root of the module.
- ***class*** – optional name of the class declaration of the implementation, including any namespace definition. If the header only contains one class then this class does not need to be specified.

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

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

Implementation declaration header file.

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

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

Implementation.

```
#include "LoanServiceImpl.h"

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

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

```
<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/0.9">
   <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"?>
<module  xmlns="http://www.osoa.org/xmlns/sca/0.9"

   name="LoanModule" >

   …

   <component name="LoanService">
         <implementation.cpp dll="loan.dll" header="LoanServiceImpl.h”/>
   </component>

   …

</module>
```

1.3. Basic Client Model

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

1.3.1. Accessing Services from Component Implementations

A service can get access a service using a component context. To access a service using the module context two things have to be done:

1. the current component context must be retrieved
2. a method has to be called on the component context.

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

```
namespace osoa {
    namespace sca {

       class ComponentContext {
       public:
          static ComponentContext getCurrent();
          virtual void * getService(const char *referenceName);
          …
       }
    }
}
```

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;

   …

   ComponentContext context = ComponentContext::getCurrent();

   CustomerService* service =
               (CustomerService* )context.getService("customerService");

   short rating = service->getCreditRating(customerNumber);

}
```

1.3.2. Accessing Services from non-SCA component implementations

The following sections describe how non-SCA components that are part of an SCA module get access to services.

1.3.2.1. Using Module Context

Non-SCA code that is part of an SCA module can use the ModuleContext in their implementations in order to find services. Non-SCA code is considered to be part of an SCA module if it has been loaded in the same process and memory as an SCA module. One SCA module will be the default module which is the first module to be loaded by the SCA runtime, and this module will be used if an explicit module is not set.

The following snippet shows the ModuleContext C++ interface.

```
namespace osoa {
   namespace sca {

         class ModuleContext {
         public:
               static ModuleContext getCurrent();

               virtual void * locateService(const char *serviceName);
               …
         };
   }
}
```

The non-SCA component would include a line like the following in its implementation to get access to the module context:

```
ModuleContext moduleContext = ModuleContext::getCurrent();
```

Once the module context is available, the required service can be found using the locateService() method of the module context.

1.3.3. Calling Service Methods

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

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

1.4. Error Handling

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

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

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

- ***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.

1.5. C++ API

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

1.5.1. Module Context

The following snippet shows the ModuleContext interface.

```
class ModuleContext {
public:
   static ModuleContext getCurrent();
   virtual void * locateService(const char *serviceName);
   virtual const char * getName();
   virtual const char * getURI();
   virtual DataObjectPtr getMetaData();
};
```

The ModuleContext C++ interface has the following methods:

- ***getCurrent()*** – a static method returning context for the current module.
- ***getName()*** – returns the name of the module.
- ***getURI()*** – returns the absolute URI of the module component.
- ***getMetaData()*** – returns an SDO data object pointer that represents the Module meta data object. The module data object and all the other model data objects are created from the SCA XML schema using the SDO 2.0 [1] XML schema to C++ mapping rules.
- ***locateService()*** – returns a pointer to an object implementing the interface defined for the named service. Input to the method is the name of a service that is part of the same SCA module. The service name must be the <component-name>/<service-name> where the service-name is optional if the component has only one service defined

1.5.2. Component Context

The following snippet shows the ComponentContext interface.

```
class ComponentContext {
public:
   static ComponentContext getCurrent();
   virtual void * getService(const char *referenceName);
   virtual ServiceList getServices(const char *referenceName);
   virtual DataObjectPtr getProperties();
   virtual DataFactoryPtr getDataFactory();
};
```

The ComponentContext C++ interface has the following methods:

- ***getCurrent()*** – returns the ComponentContext for the current 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. An exception 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.
- ***getProperties()*** – Returns an SDO from which all the properties defined in the componentType file can be retrieved.
- ***getDataFactory()*** – Returns an SDO DataFactory which can be used to create data objects.

1.5.3. ServiceList

The ServiceList class holds a list of services that can individually be cast to the business interface.

```
class ServiceList {
public:
   virtual void * operator[](int position);
   virtual unsigned int size(void);

};
```

- The ServiceList class provides the [] operator to allow access to the list by index.
- ***size()*** – The number of services in the list.

1.6. Proposed Future Additions to the Specification

The current version of the specification is acknowledged to be incomplete and there are a number of areas where future versions of the specification will be expanded to cover additional capabilities. Community feedback is welcomed for any of these topics. Some of these capabilities include:

- An asynchronous programming model.
- A dynamic invocation interface.
- The use of annotations in header files to describe SCA artifacts without needing to include the details in side files.
- The use of injection to set properties and references of a component implementation.
- A C++ to WSDL mapping.

2. Appendix 1

2.1. Packaging and Deployment

2.1.1. Module Packaging

The physical realization of an SCA module is a folder in a file system which is named like the module and which must contain one and only one sca.module file at its root. The following shows the MyValueModule just after creation in a file system.

```
MyValueModule/
      sca.module
```

Besides the sca.module file the module contains artifacts that define the implementations of components, and that define the bindings of external services and entry points. Samples for 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 module, whereby programmers have the freedom to construct a folder structure that best fits the needs of their project. The following shows the complete MyValueModule folder file structure in a file system.

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

Note that the folder structure is not architected, other than the sca.module (and any .fragment files) must be at the root of the folder structure.

Note that it is not possible in a module to mix component implementations written in C++ with those written in any other language.

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

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

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

An SCA module is the smallest unit of deployment in an ***SCA system***. Deployed modules are used and configured in the SCA system by ***SCA subsystem*** configurations.

2.1.2. Subsystem Packaging

The physical realization of an SCA subsystem can be a ***folder in a file system*** or which is named like the subsystem and which must contain one and only one ***sca.subsystem*** file at its root. The following shows the MyValueSubsystem in a file system.

```
MyValueSubsystem/
      sca.subsystem
```

Subsystems contain module components that use and configure instances of SCA modules deployed in the SCA system.

2.1.3. Deployment

SCA Modules and ***SCA subsystems*** get deployed in SCA systems. An SCA system will have two architected folders one named ***modules*** that contains the deployed SCA modules, and the other named ***subsystems*** that contain the deployed SCA subsystems.

The modules folder contains one subfolder per module that either can contain the module contents in expanded or archive form. Similarly the subsystems folder contains one subfolder per subsystem that either can contains the subsystem contents in expanded or archive form.

See SCA Assembly model [2] for more description of the deployment model.

During deployment, the shared libraries packaged in an SCA Module may be moved to a different location on the target system to be accessible at runtime. Also any shared libraries that are dependencies on the libraries packaged in the SCA Module must be accessible by the normal library loading rules for the target environment (for example, included on the PATH environment variable on Windows, or in /usr/lib on Linux).

3. Appendix 2

3.1. Types Supported in Service Interfaces

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

3.1.1. Local service

For a local service the types that are supported are:

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

3.1.2. Remotable service

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

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

Not supported:

- Pointers.
- References.

4. Appendix 3

4.1. Restrictions on C++ header files

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

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

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

- Macros
- inline methods
- friend classes

5. Appendix 4

5.1. XML Schemas

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

5.1.1. sca-interface-cpp.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.commonj.org/xmlns/sca/v0.0.1/"
        xmlns:sca="http://www.commonj.org/xmlns/sca/v0.0.1/"
        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">
                     <attribute name="header" type="Name" use="required"/>
                     <attribute name="class" type="Name" use="required"/>
                     <attribute name="scope" type="sca:CPPScope" use="optional"/>
                     <attribute name="remotable" type="boolean" use="optional"/>
                     <anyAttribute namespace="##any" processContents="lax"/>
               </extension>
         </complexContent>
   </complexType>

   <simpleType name="CPPScope">
         <restriction base="string">
               <enumeration value="stateless"/>
               <enumeration value="module"/>
         </restriction>
   </simpleType>

</schema>
```

5.1.2. sca-implementation-cpp.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright BEA Systems Inc. and IBM Corporation 2005 -->
<schema  xmlns="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://www.osoa.org/xmlns/sca/0.9"
         xmlns:sca="http://www.osoa.org/xmlns/sca/0.9"
         xmlns:sdo="commonj.sdo/XML"
         elementFormDefault="qualified">

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

   <element name="implementation.cpp" type="sca:CPPImplementation"
          substitutionGroup="sca:implementation" sdo:name="implementationCpp"/>
   <complexType name="CPPImplementation">
      <complexContent>
               <extension base="sca:Implementation">
                     <sequence>
                           <any namespace="##other" processContents="lax"
                                       minOccurs="0" maxOccurs="unbounded"/>
                     </sequence>
                     <attribute name="dll" type="NCName" use="required"/>
                     <attribute name="header" type="NCName" use="required"/>
                     <attribute name="class" type="Name" use="optional"/>
                     <anyAttribute namespace="##any" processContents="lax"/>
               </extension>
         </complexContent>
   </complexType>
</schema>
```

6. References

[1] SDO 2.0 Specification

Any one of:

- http://dev2dev.bea.com/technologies/commonj/index.jsp
- http://www.ibm.com/developerworks/library/specification/ws-sdo/
- http://oracle.com/technology/webservices/sca
- https://www.sdn.sap.com/
- http://www.xcalia/xdn/specs/sdo
- http:/www.sybase.com

[2] SCA Assembly Specification

Any one of:

- http://dev2dev.bea.com/technologies/commonj/index.jsp
- http://www.ibm.com/developerworks/library/specification/ws-sca/
- http://www.iona.com/devcenter/sca/
- http://oracle.com/technology/webservices/sca
- https://www.sdn.sap.com/
- http://www.sybase.com