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

Committee Draft 01

20 March 2008

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

Previous Version:

N/A

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

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

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

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

Related work:
This specification replaces or supercedes:

- OSOA SCA C Client and Implementation V1.00

This specification is related to:

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

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

Abstract:

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

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

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

Status:

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

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

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

The non-normative errata page for this specification is located at http://www.oasis-open.org/committees/sca-c-cpp/.

Notices

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

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

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

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

1.1 Terminology

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

1.2 Normative References

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

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

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

[SDO21] B. Aupperle, et al., *Service Data Objects For C Specification*, http://www.osoa.org/download/attachments/36/SDO_Specification_C_V2.1.pdf, SDO 2.1, September 2007.

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

[WSDL20] R. Chinnici, et al., *Web Service Description Language (WSDL) 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

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 set of C functions (a ***C implementation***) provides one or more services.

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

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

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

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

Service interface.

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

Implementation.

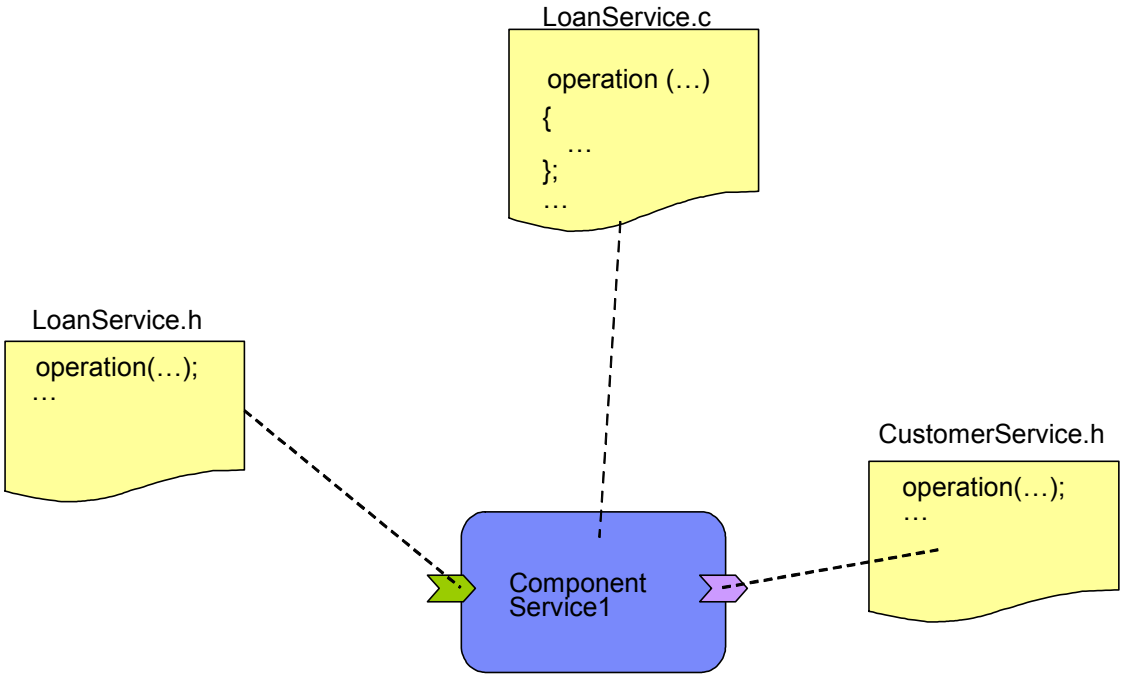
```
#include "LoanService.h"

char approveLoan(long customerNumber, 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.c 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:

```
<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
   <service name="LoanService">
          <interface.c 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 a remotable service that allows pass by reference MUST NOT alter its input data during or after the invocation, and MUST NOT modify return data after invocation. The *allowsPassByReference=true* attribute on the implementation.c element of a remotable service is used to either declare that calls to the whole interface allows pass by reference. , this attribute can be used on a specific function.

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.c element indicates that the service contract is conversational as described in the Assembly Specification.

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 and 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.c element.

When a scope is not specified in an implementation file, 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.c module="loan" scope="composite"/>
</component>
```

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

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

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

   <function … />*

</implementation.c>
```

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

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

Implementing a Service with a Program).

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 components, there is no implied correlation between service requests.

Lifecycle functions are not defined for stateless implementations.

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, i.e. a copy of the binary implementing the component is loaded into a new memory space for each remote request and all corresponding local requests are processed by this remote request-specific instance. 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.

Request scope supports both init=”true” and destroy=”true” functions.

2.3.3 Composite scope

All service requests are dispatched to the same implementation instance for the lifetime of the containing composite, i.e. the binary implementing the component is loaded into memory once and all requests are processed by this single instance. 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.

Composite scope supports both init=”true” and destroy=”true” functions.

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 Operations 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 having a stateless (or other) scope could implement a conversational service.

Lifecycle functions are not defined for conversational implementations.

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. The C component can retrieve properties values using the SCAProperty<PropertyType>() functions, for example SCAPropertyInt() to access an Int type property..

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

```
#include "SCA.h"

void clientFunction()
{

   …

   int32_t loanRating;
   int compCode, reason;

…

   SCAPropertyInt(L"maxLoanValue", &loanRating, &compCode, &reason);

   …

}
```

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

2.5 Component Type and Component

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

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 a C service interface and a C implementation of a service.

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

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

Implementation.

```
#include "LoanService.h"

char approveLoan(long customerNumber, 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.c header="LoanService.h" componentType=”LoanService” />
   </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.c module="loan"/>
   </component>

   …

</composite>
```

2.5.1 Interface.c

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.c schema snippet -->
<interface.c xmlns=”http://docs.oasis-open.org/ns/opencsa/sca/200712“
      header="NCName" remotable="boolean"? callbackHeader=”Name”? >

   <function … />*
   <callbackFunction … />*

</interface.c>
```

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

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

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

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

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

2.5.2 Function and CallbackFunction

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

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

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

   <function name="NCName" oneWay="Boolean"? endsConversation="Boolean"?
         input="NCName"? output="NCNAME"? />*
   <callbackFunction name="NCName" oneWay="Boolean"?
         endsConversation="Boolean"? input="NCName"? output="NCName"? />*

</interface.c>
```

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

- ***name : NCName (1..1) –*** name of the function being decorated or included in the interface.
- ***oneWay : boolean (0..1) –*** see **Error! Reference source not found.**
- ***endsConversation ; boolean (0..1) –*** see Operations that End the Conversation

input : NCNAME (0..1) – If the interface is implemented by a program (seeSome methods of an implementation have operational characteristics that need to be identified. This is done using a method child element of implementation.c

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

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

   <function … />*

</implementation.c>
```

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

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

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

2.5.3 Implementation.c

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.c schema snippet -->
<implementation.c xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
      module="NCName" library="boolean"? location="Name"
      scope="scope"? componentType="Name" allowsPassByReference=Boolean"?
      eagerInit="boolean"? init="boolean"? destroy="boolean"?
      conversationMaxAge="string"? conversationMaxIdle="string"?
      conversationSinglePrincipal="Boolean"? >

   <function … />*

</implementation.c>
```

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

- ***module : NCName (1..1) –*** name of the binary executable for the service component. This is the root name of the module. A runtime specific prefix and/or suffix MAY be added.
- ***library : boolean (0..1) –***indicates whether the service is implemented as a library or a program. The default is library.
- ***location : Name (0..1) –*** location of the module as either a full path, its equivalent, or a relative path from the composite root.
- ***scope : CImplementationScope (0..1) –*** indicates the scope of the component implementation. The default is stateless.
- ***componentType : Name (1..1) –*** name of the componentType file. A ".componentType" extention will be appended. An optional path to the componentType file which is relative to the root of the composite MAY be included.
- ***allowsPassByReference : boolean (0..1) –*** indicates the service allows pass by reference data exchange semantics on calls to it.
- ***eagerInit : boolean (0..1) –*** indicates a composite scoped implementation should be initialized when it is loaded.
- ***init : boolean (0..1) –*** indicates program should be called with an initialize flag to initialize the implementation.
- ***destroy : boolean (0..1) –*** indicates should be called with an destroy flag to to cleanup the implementation.
- ***conversationMaxAge : string (0..1) –*** see Error! Reference source not found..
- ***conversationMaxIdle : string (0..1) –*** see Error! Reference source not found..
- ***conversationSinglePrincipal : boolean (0..1) –*** see Error! Reference source not found..

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

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

2.5.4 Implementation Function

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

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

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

   <function … />*

</implementation.c>
```

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

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

2.6 Implementing a Service with a Program

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

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

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

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

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

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

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

```
#include "SCA.h"
#include "myInterface.h"
main () {
    wchar_t myService [255];
    wchar_t myOperation [255];
    int compCode, reason;
    struct FirstInputMsg myFirstIn;
    struct FirstOutputMsg myFirstOut;


    SCAService(myService, &compCode, &reason);

    SCAOperation(myOperation, &compCode, &reason);

    if (wstrcmp(myOperation,L"myFirstOperation")==0){
        SCAMessageIn(myService, myOperation,
                     sizeof(struct FirstInputMsg), (void *)&myFirstIn,
                     &compCode, &reason);
        …
        SCAMessageOut(myService, myOperation,
                      sizeof(struct FirstOutputMsg),(void *)&myFirstOut,
                      &compCode, &reason);
    }
    else
       {
           …
       }
}
```

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 operations of these services.

3.1 Accessing Services from Component Implementations

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

The following shows the function used for this.

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

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

```
#include "SCA.h"

void clientFunction()
{

   SCAREF serviceToken;
   int compCode, reason;
   long custNum = 1234;
   short rating;

  …
   SCALocate(L”customerService”, &serviceToken, &compCode, &reason);
   SCAInvoke(serviceToken, L“getCreditRating”, sizeof(custNum),
             (void *)&custNum, sizeof(rating), (void *)&rating,
             &compCode, &reason);

}
```

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

```
SCAREF *tokens;
int num_targets;
...
myFunction(...) {
   int compCode, reason;
   ...
   SCALocateMultiple(L"myReference", &tokens, &num_targets, &compCode,
                     &reason);
   for (i = 0; i < num_targets; i++)
   {
      // set up callback function
      SCASetCallback(tokens[i], L"myCallback", pfn, &compCode, &reason);
      // set up arguments
      SCAInvoke(tokens[i], L"myOperation", sizeof(inputMsg),
```

```
                    (void *)&inputMsg, 0, NULL, &compCode, &reason);
    };
};
```

3.2 Accessing Services from non-SCA component implementations

Non-SCA components can access component services by obtaining an SCAREF 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 SCAREF is not defined by this specification.

3.3 Calling Service Operations

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

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

4 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 operation 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.

4.1 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 "SCA.h"
#include "LoanApplicationClient.h"
#include "LoanService.h"
#include "LoanApplication.h"

SCAREF serviceToken;

void clientFunction(struct LoanApplication loanApp, int term)
{

   long customerNumber = 1234;
      int compCode, reason;

   SCALocate(L"loanService", &serviceToken, &compCode, &reason);

   SCAInvoke(serviceToken, L"apply", sizeof(loanApp), (void *)&loanApp,
             0, NULL, &compCode, &reason);
   SCAInvoke(serviceToken, L"lockCurrentRate", sizeof(term), (void *)&term,
             0,  NULL, &compCode, &reason);

}

bool isApproved()
{
      int status;
      int compCode, reason;

      SCAInvoke(serviceToken, L"getLoanStatus", 0, NULL,
                sizeof(status), (void *)&status, &compCode, &reason);
   return (status == 1);
}
```

4.2 Conversational Service Provider

A implementation which provides a service with a conversational interface can have any scope. In particular, it is not necessary for the implementation 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 implementation, with state data held in instance variables. However, for implementations with other scopes, when an operation of a conversational interface is executing, the SCAGetConversationID() returns the conversation ID of the conversation. The conversation ID can be used by the implementation as an index to store and to look up state data associated with the conversation, using some suitable storage mechanism.

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

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

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 timeouts are defined by the implementation of the SCA runtime, however it chooses to do so.

Here is an example implementation definition of a conversational service.

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

4.3 Operations that End the Conversation

An operation of a conversation scoped implementation can be marked with an endsConversation=”true” attribute. This means that once this operation has been called, no further operation is 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 an operation 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 an operation is called after the conversation completes, the SCA_CONVERSATION_ENDED reason code is returned. This can also occur if there is a race condition between the client and the service provider calling their respective endsConversation operations. Calling the SCAEndConversation() function also ends a conversation.

The following is an example implementation with a function 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.c header="LoanService.h" remotable=”true”
            requires=”conversational”>
         <function name=”cancelApplication” endsConversation=”true” />
      </interface.c>
   </service>
</componentType>
```

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

4.4 Conversation Lifetime Summary

Starting conversations

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

- A service is located using SCALocate().

Continuing conversations

The client can continue an existing conversation, by:

Holding the service token that was created when the conversation started

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 an operation marked endsConversation=”true” on the callback interface.
- The service's conversation lifetime timeout occurs.
- The client calls SCAEndConversation().
- The client calls SCASetConversationID() which implicitly ends any active conversation.

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

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

4.5 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 SCASetConversationID().

```
wchar_t conversationID[] = L”myID”;
SCAREF serviceToken;
int compCode, reason;

SCALocate(L”MyService”, &serviceToken, &compCode, &reason);
SCASetConversationID(serviceToken, conversationID, &compCode, &reason);
```

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.

4.6 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 SCAGetConversationID().

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

5 Asynchronous Programming

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

5.1 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 function that returns "void" and only uses by-value parameters can be marked by using the **oneWay=true** attribute in the interface definition of the service. This means that the function is non-blocking and communication with 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() function declared as a one-way operation:

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

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

5.2 Callbacks

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

A callback interface is declared by the *callbackHeader* and *callbackFunctions* 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.c header="MyService.h" callbackHeader="MyServiceCallback.h"/>
   </service>
</componentType>
```

5.2.1 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 SCAGetCallback() function.

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

```
#include "SCA.h"
#include "MyService.h"
#include "MyServiceCallback.h"

void someFunction(int arg)
{
      SCAREF serviceRef;
      int compCode, reason;
   …
/* do some processing… */
   SCAGetCallback(L"", &serviceRef, &compCode, &reason);
   SCACallback(serviceRef, L"receiveResult",
               sizeof (result), (void *)&result, 0, NULL,
               &compCode, &reason);
}
```

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

```
#include "SCA.h"
#include "MyService.h"
#include "MyServiceCallback.h"

void clientFunction(int arg)
{
   SCAREF serviceToken;
   int compCode, reason;

   /* invoke the service */
SCALocate(L"MyService", &serviceToken, &compCode, &reason);
SCAInvoke(serviceToken, L"someFunction", sizeof(arg), (void *)&arg, 0, NULL,
          &compCode, &reason);
}

receiveResult(int result)
{
   /* code to process result */
}
```

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.

5.2.2 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 operations 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 operation) is a callback ID that is passed with requests to the service and is guaranteed to be returned with any callback.

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

```
SCAREF serviceToken;
int compCode, reason;

void clientFunction(int arg)
{
   /* invoke the service */
SCALocate(L"MyService, &serviceToken, &comCode, &reason);
SCASetCallbackID(serviceToken, L"1234" , &compCode, &reason);
SCAInvoke(serviceToken, L"someFunction", sizeof(arg), (void *)&arg,
          0, NULL, &compCode, &reason);
}


receiveResult(int result)
{
   wchar_t *myID;
      int compCode, reason;

SCAGetCallbackID(serviceToken, &myID, &compCode, &reason);

   /* code to process result */
}
```

5.2.3 Implementing Multiple Bidirectional Interfaces

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

5.2.4 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 using SCASetCallbackID(). This can be used even 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 operation 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.

6 Error Handling

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

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

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

```
/* Completion Codes */
#define SCACC_OK             0
#define SCACC_WARNING        1
#define SCACC_FAULT          2
#define SCACC_ERROR          3

/* Reason Codes */
#define SCA_SERVICE_UNAVAILABLE    1
#define SCA_CONVERSATION_ENDED     2
#define SCA_NO_REGISTERED_CALLBACK 3
#define SCA_MULTIPLE_SERVICES      4
#define SCA_DATA_TRUNCATED         5
```

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

7 C API

7.1 Synchronous Programming

The following shows the C interface declarations for synchronous programming.

```
typedef void *SCAREF;

void SCALocate(wchar_t *referenceName,
               SCAREF *referenceToken,
               int *compCode,
               int *reason);

void SCALocateMultiple(wchar_t *referenceName,
                       SCAREF **referenceTokens,
                       int *num_targets,
                       int *CompCode,
                       int *Reason);

void SCAInvoke(SCAREF referenceToken,
               wchar_t *operationName,
               int inputMsgLen,
               void *inputMsg,
               int outputMsgLen,
               void *outputMsg,
               int *compCode,
               int *reason);

void SCAPropertyBoolean(wchar_t *propertyName,
                        char *value,
                        int *compCode,
                        int *reason);

void SCAPropertyByte(wchar_t *propertyName,
                     int8_t *value,
                     int *compCode,
                     int *reason);

void SCAPropertyBytes(wchar_t *propertyName,
                      int8_t **value,
                      int *size,
                      int *compCode,
                      int *reason);

void SCAPropertyChar(wchar_t *propertyName,
                     wchar_t *value,
                     int *compCode,
                     int *reason);

void SCAPropertyChars(wchar_t *propertyName,
                      wchar_t **value,
                      int *size,
                      int *compCode,
                      int *reason);

void SCAPropertyCChar(wchar_t *propertyName,
                      char *value,
                      int *compCode,
                      int *reason);
```

```
void SCAPropertyCChars(wchar_t *propertyName,
                       char **value,
                       int *size,
                       int *compCode,
                       int *reason);

void SCAPropertyShort(wchar_t *propertyName,
                      int16_t *value,
                      int *compCode,
                      int *reason);

void SCAPropertyInt(wchar_t *propertyName,
                    int32_t *value,
                    int *compCode,
                    int *reason);

void SCAPropertyLong(wchar_t *propertyName,
                     int64_t *value,
                     int *compCode,
                     int *reason);

void SCAPropertyFloat(wchar_t *propertyName,
                      float *value,
                      int *compCode,
                      int *reason);

void SCAPropertyDouble(wchar_t *propertyName,
                       double *value,
                       int *compCode,
                       int *reason);

void SCAPropertyString(wchar_t *propertyName,
                       wchar_t **value,
                       int *compCode,
                       int *reason);

void SCAPropertyCString(wchar_t *propertyName,
                       char **value,
                       int *compCode,
                       int *reason);

void SCAPropertyDate(wchar_t *propertyName,
                     struct tm *value,
                     int *compCode,
                     int *reason);

void SCAGetFaultMessage(SCAREF referenceToken,
                        int bufferLen,
                        char *buffer,
                        int *compCode,
                        int *reason);

void SCASetFaultMessage(wchar_t *serviceName,
                        wchar_t *operationName,
                        int bufferLen,
                        char *buffer,
                        int *compCode,
                        int *reason);
```

The C Synchronous Programming interface has the following functions:

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

- ***SCALocateMultiple()*** – provides a pointer to an array of tokens, one for each target of a reference with multiple targets.
- ***SCAInvoke()*** – invokes an operation of a service.
- ***SCAProperty<PropertyType>()*** – provides the value of the specified property. This API is available for Boolean, Byte, Bytes, Char, Chars, CChar, CChars, Short, Int, Long, Float, Double, String, CString, and Date. The Char, Chars, and String variants return wchar_t based data while the CChar, CChars, and CString variants return char base data.
  The Bytes, Chars, and CChars variants return a buffer of data; a size parameter is used to specify the maximum size of the buffer and is updated to return the actual size of the data. The String and CString variants return a null terminated string; a size parameter is used to specify the maximum length of the string that can be returned. If the value for the parameter is too large to fit in the provided space, CompCode is set to SCACC_WARNING, Reason is set to SCA_DATA_TRUNCATED, and the size parameter is set to the space required to hold the entire value.
- ***SCAGetFaultMessage()*** – retrieve a fault message for a service operation.
- ***SCASetFaultMessage()*** – set a fault message for a service operation.

7.2 Program-Based Implemenation Support

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

```
void SCAService(wchar_t *serviceName, int *compCode, int *reason);

void SCAOperation(wchar_t *operationName, int *compCode, int *reason);

void SCAMessageIn(wchar_t *serviceName,
                  wchar_t *operationName,
                  int bufferLen,
                  void *buffer,
                  int *compCode,
                  int *reason);

void SCAMessageOut(wchar_t *serviceName,
                   wchar_t *operationName,
                   int bufferLen,
                   void *buffer,
                   int *CompCode,
                   int *Reason);
```

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

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

7.3 Asynchronous Programming

The following shows the C interface declarations for asynchronous programming.

```
void SCAGetCallback(wchar_t *serviceName,
                    SCAREF *referenceToken,
                    int *compCode,
                    int *reason);
```

```
void SCACallback(SCAREF referenceToken,
                 wchar_t *operationName,
                 int inputMsgLen,
                 void *inputMsg,
                 int outputMsgLen,
                 void *outputMsg,
                 int *compCode,
                 int *reason);

void SCASetCallbackID(SCAREF referenceToken,
                      wchar_t *id,
                      int *compCode,
                      int *reason);

void SCAGetCallbackID(SCAREF referenceToken,
                      wchar_t **id,
                      int *compCode,
                      int *reason);
```

The C Asynchronous Programming interface has the following functions:

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

7.4 Conversational Services

The following shows the C interface declarations for conversations.

```
void SCAGetConversationID(wchar_t *serviceName,
                          wchar_t **id,
                          int *compCode,
                          int *reason);

void SCASetConversationID(SCAREF referenceToken,
                          wchar_t *id,
                          int *compCode,
                          int *reason);

void SCAEndConversation(SCAREF referenceToken,
                        int *compCode,
                        int *reason);
```

The C Conversational Services interface has the following functions:

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

8 C to WSDL Mapping

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

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

8.1 Parameter and Return Type mappings

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

8.1.1 Built-in type mappings

C 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

[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.

char* or char array	A null-terminated UTF-8 encoded string	xsd:string
wchar_t* or wchar_t array	A null-terminated UTF-16 or UTF-32 encoded string[2]	xsd:string
struct tm		xsd:dateTime

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

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

would generate a schema like:

```
<xsd:element name="myFunction">
  <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"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="myFunctionResponse">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="myFunctionResponseData" type="xsd:int"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

8.1.2 Binary data mapping

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

8.1.3 Array mapping

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

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

would generate a schema like:

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

[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.

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

8.1.4 Multi-dimensional array mapping

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

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

would generate a schema like:

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

8.1.5 Pointer mapping

A C function prototype that uses the ‘pass-by-pointer’ style, defining parameters that are either pointers, is not meaningful when applied to web services, which rely on serialized data. A C function prototype that uses 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 function.

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

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

would generate a schema like:

```
<xsd:element name="myFunction">
  <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"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

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

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

8.1.6 Struct mapping

C `struct`s that contain types that can be mapped, are themselves mapped to complex types. For example, a function such as:

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

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

```
struct DataStruct {
  char *name;
  double value;
};
```

would convert to a schema like:

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

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

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

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

```
  double value;
};
```

would convert to a schema like:

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

<xsd:complexType name="DataStruct">
  <xsd:sequence>
    <xsd:element name="name">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:maxLength value="255"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
    <xsd:element name="value" type="xsd:double"/>
  </xsd:sequence>
</xsd:complexType>
```

8.1.7 Enum mapping

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

For example, a function such as:

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

with the ParameterType type defined as an `enum`:

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

would convert to a schema like:

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

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

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

```
enum ParameterType {
   UNSET = 'u',
   TYPEA = 'A',
   TYPEB = 'B',
   TYPEC = 'C'
};
```

would 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>
```

8.1.8 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.

8.1.9 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.

8.1.10 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.

8.1.11 Nesting types

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

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

with types defined as follows:

```
struct DataStruct {
  char name[30];
  double values[20];
  ParameterType type;
};

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

would convert to a schema like:

```
<xsd:element name="myFunction">
  <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">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:maxLength value="29"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
    <xsd:element name="values" type="xsd:double" minOccurs=20
maxOccurs=20/>
    <xsd:element name="type" type=" ParameterType"/>
  </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>
```

8.1.12 SDO mapping

C function prototypes that use dynamic SDO DATAOBJECT 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 function prototype defined in a header such as:

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

would generate a schema like:

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

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

8.1.13 void * mapping

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

8.1.14 Included types

An implementation MUST completely map a C interface to WSDL, including types (`struct`s, `enum`s, 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) MUST be mappable to a schema element via the rules in this document.

8.2 Header mapping

Unless otherwise specified with annotations, all of the functions in a C header map 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 functions defined in the C header. 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 (MyService.h) containing:

```
int myFunction(char *data);
double myOtherFunction(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="myFunction">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="data" type="xsd:string"/>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="myFunctionResponse">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="myFunctionResponseData"
             type="xsd:short"/>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name="myOtherFunction">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="otherData" type="xsd:double"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="myOtherFunctionResponse">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="myOtherFunctionResponseData"
             type="xsd:double"/>
      </xsd:complexType>
    </xsd:element>

  </xsd:schema>
</types>

<message name="myFunctionRequestMsg">
  <part name="body" element="tns:myFunction"/>
</message>
<message name="myFunctionResponseMsg">
  <part name="body" element="tns:myFunctionResponse"/>
</message>

<message name="myOtherFunctionRequestMsg">
  <part name="body" element="tns:myOtherFunction"/>
</message>
<message name="myOtherFunctionResponseMsg">
  <part name="body" element="tns:myOtherFunctionResponse"/>
</message>

<portType name="MyServicePortType">
  <operation name="myFunction">
    <input message="tns:myFunctionRequestMsg"/>
    <output message="tns:myFunctionResponseMsg"/>
  </operation>
  <operation name="myOtherFunction">
    <input message="tns:myOtherFunctionRequestMsg"/>
    <output message="tns:myOtherFunctionResponseMsg"/>
  </operation>
</portType>
```

```
  <binding name="MyServiceBinding" type="tns:MyService">
    <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="myFunction">
      <soap:operation soapAction="MyService#myFunction"/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
    <operation name="myOtherFunction">
      <soap:operation soapAction="MyService#myOtherFunction"/>
      <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 annotations are used to define multiple interfaces in a single C header file, then each interface maps to a unique Binding and PortType element and the service element contains multiple port elements.

8.3 Function mapping

8.3.1 Unnamed parameters

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

8.3.2 The return type

The return type in C functions 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 function prototype above, the response data will be returned using the following schema:

```
<xsd:element name="myFunctionResponse">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="myFunctionResponseData" type="xsd:short"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

8.3.3 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 myFunction(char *name, double value);
```

would generate a schema like:

```
<xsd:element name="myFunctionRequestMsg">
  <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="myFunctionResponseMsg">
  <xsd:complexType/>
</xsd:element>
```

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

```
<portType name="MyServicePortType">
  <operation name="myFunction">
    <input message="tns:myFunctionRequestMsg"/>
    <output message="tns:myFunctionResponseMsg"/>
  </operation>
</portType>

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

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

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

the following schema would be generated:

```
<xsd:element name="myFunctionRequestMsg">
  <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="myFunction">
    <input message="tns:myFunctionRequestMsg"/>
  </operation>
</portType>

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

8.3.4 No Parameters Specified

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

9 WSDL to C Mapping

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

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

- /definitions/portType
- /definitions/operation
- /definitions/message
- /definitions/types

The physical contract is defined by these XML elements:

- /definitions/service
- /definitions/binding
- /definitions/port

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

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

9.1 Type Mapping

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

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

9.1.1 Simple type mapping

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

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

1667

Notes:

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

Example:

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

maps to:

```
wchar_t *myString;
/* this points to a dymically allocated buffer with the data */

<xsd:simpleType name="boundedString25">
  <xsd:restriction base="xsd:string">
    <xsd:length value="25"/>
  </xsd:restriction>
</xsd:simpletype>
…
<xsd:element name="myString" type="boundedString25"/>
maps to:
wchar_t myString[26];
```

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

Examples:

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

maps to:

```
char *myData;
/* this points to a dymically allocated buffer with the data */
```

```
<xsd:simpleType name="boundedData25">
  <xsd:restriction base="xsd:hexBinary">
    <xsd:length value="25"/>
  </xsd:restriction>
</xsd:simpletype>
…
<xsd:element name="myData" type="boundedData25"/>
```

maps to:

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

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

Examples:

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

maps to:

```
char *myInteger;
/* this points to a dymically allocated buffer with the data */
```

```
<xsd:element name=”myBoundedInteger”>
  <xsd:simpleType>
    <xsd:restriction base=”xsd:integer”>
      <xsd:minInclusive value=”-20000”/>
      <xsd:maxInclusive value=”20000”/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

maps to:

```
long myBoundedInteger;
```

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

maps to:

```
char *myDecimal;
/* this points to a dymically allocated buffer with the data */
```

```
<xsd:element name=”myBoundedDecimal”>
  <xsd:simpleType>
    <xsd:restriction base=”xsd:decimal”>
      <xsd:totalDigits value=”10”/>
      <xsd:fractionDigits value=”2”/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

maps to:

```
double myBoundedDecimal;
```

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

Examples:

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

maps to:

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

```
<xsd:simpleType name=”lineNumList”>
  <xsd:list itemType=”xsd:int”/>
</xsd:simpleType>
<xsd:simpleType name=”lineNumList6”>
  <xsd:restriction base="lineNumList ">
    <xsd:minLength value="1"/>
    <xsd:maxLength value=”6”/>
  </xsd:restriction>
```

```
</xsd:simpletype>
…
<xsd:element name=”lineNums” type=”lineNumList6”/>
```

maps to:

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

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

Examples:

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

maps to:

```
size_t dates_num;
struct tm *date;
/* this points to a dynamically allocated array of struct tm’s */
```

```
<xsd:simpleType name=”lineNumList”>
  <xsd:list itemType=”xsd:int”/>
</xsd:simpleType>
…
<xsd:element name=”lineNums” type=”lineNumList”/>
```

maps to:

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

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

9.1.2 Complex type and group mapping

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

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

Example:

```
<xsd:complexType name=”myType”>
  <xsd:sequence>
    <xsd:element name="name">
      <xsd:simpleType>
```

```
        <xsd:restriction base="xsd:string">
          <xsd:length value=”25”/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
    <xsd:element name="idList" type="xsd:int"
                 minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="value" type="xsd:double"/>
  </xsd:sequence>
</xsd:complexType>
```

maps to:

```
struct myType {
  wchar_t name[26];
  size_t idList_num;
  long *idList;
  /* this points to a dynamically allocated array of longs */
  double value;
};
```

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

Example:

```
<xsd:element name=”myVariable”>
  <xsd:complexType name=”myType”>
    <xsd:all>
      <xsd:element name="name">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:length value=”25”/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element name="idList" type="xsd:int"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="value" type="xsd:double"/>
    </xsd:all>
  </xsd:complexType>
</xsd:element>
```

maps to:

```
struct myType {
  wchar_t name[26];
  size_t idList_num;
  long *idList;
  /* this points to a dynamically allocated array of longs */
  double value;
} *pmyVariable, myVariable;
```

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

Example:

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

maps to:

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

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

9.2 Message Mapping

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

9.3 Part Mapping

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

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

9.4 Operation Mapping

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

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

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

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

9.4.1 Operation Elements <input>, <output> and <fault>

Operations fall into one of two WSDL groups:

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

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

- wsdl:tParam, used for <input> and <output> message types, and which requires a message attribute (/definitions/portType/operation/input[@message] and /definitions/portType/operation/output[@message]) to be specified. An optional name attribute (/definitions/portType/operation/input[@name] and /definitions/portType/operation/output[@name]) might be specified. The message attribute contributes to the parameter list of the operation through its parts, while the name attribute if present is ignored.
- wsdl:tFault, used for the <fault> message type, which require both a message (/definitions/portType/operation/fault[@message]) and name (/definitions/portType/operation/ fault[@name]) attribute to be specified.

The details of the C mapping are discussed in the next section.

9.4.1.1 C Mapping of <operation> Elements

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

Function Mapping Rules

The rules for mapping function are as follows:

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

Parameter Types

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

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

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

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

Example:

```
<!-- WSDL -->
...
<message name="StockIDMessage>
 <part name="exchangeSymbol" type="xs:string">
 <part name="stockSymbol" type="xs:string">
</message>
...
<portType ...>
 <operation name="GetNextStockSymbol">
```

```
  <input message="StockIDMessage">
  <output message="StockIDMessage">
 </operation>
 ...
</portType>
...
```

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

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

Example:

```
<!-- WSDL -->
...
<message name="StockIDMessage>
 <part name="exchangeSymbol" type="xs:string">
 <part name="stockSymbol" type="xs:string">
</message>
<message name="StockQuoteMessage">
 <part name="stockSymbol" type="xs:string">
 <part name="high" type="xs:decimal">
 <part name="low" type="xs:decimal">
 <part name="last" type="xs:decimal">
</message>
...
<portType ...>
 <operation name="GetStockPrice">
  <input message="StockIDMessage">
  <output message="StockQuoteMessage">
 </operation>
 ...
</portType>
...
```

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

Parameter Mapping Rules

The rules for mapping function parameters are as follows:

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

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

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

Example:

```
<!-- WSDL -->
 ...
<message name="StockIDMessage>
 <part name="exchangeSymbol" type="xs:string">
 <part name="stockSymbol" type="xs:string">
</message>
<message name="StockQuoteMessage">
 <part name="stockSymbol" type="xs:string">
 <part name="high" type="xs:float">
 <part name="low" type="xs:float">
 <part name="last" type="xs:float">
</message>
 ...
<portType ...>
 <operation name="GetStockPrice">
<input message="StockIDMessage">
<output message="StockQuoteMessage">
 </operation>
 ...
</portType>
...
```

results in the C function

```
void GetStockPrice(wchar_t *exchangeSymbol,
                   wchar_t *stockSymbol,
                   float *high,
                   float *low,
                   float *last);
```

Fault Mapping

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

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

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

The C mapping rules for user faults are as follows:

– The `struct` isderived from the `<message> <part>`s.

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

Example:

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

```
/* C */
struct FaultException {
…
builtInType partName;
…
};
```

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

```
/* C */
struct FaultException {
…
struct type partName;
…
};
```

9.4.2 One-way Operation

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

Example:

```
<!-- WSDL -->
...
<message name="ReadyMessage">
 <part name="boolFlag" type="xs:boolean"/>
</message>
...
<portType name="StockQuoteEngine">
 <operation name="StartEngine">
  <input message="tns:ReadyMessage"/>
 </operation>
</portType>
...
```

maps to:

```
/* C */
...
```

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

9.4.3 Request-Response Operation

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

Example:

```
<!-- WSDL -->
...
<message name="VoidMessage"/>
<message name="ReadyMessage">
 <part name="boolFlag" type="xs:boolean"/>
</message>
...
<portType name="StockQuoteEngine">
 <operation name="IsEngineReady">
  <input message="tns:VoidMessage"/>
  <output message="tns:ReadyMessage"/>
 </operation>
</portType>
...
```

maps to:

```
/* C */
...
void IsEngineReady(char *boolFlag);
...
```

9.5 PortType Mapping

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

Example:

```
<!-- WSDL -->
...
<portType name="TutorialA">
 ...
</portType>
<portType name="TutorialBExample">
 ...
</portType>
...
```

results in C mapping:

```
TutorialA.h
/* ... operations ... */
and
TutorialBExample.h
/* ... operations ... */
```

or

```
<wsdl>.h
/* @Service(name="TutorialA") */
/* ... operations ... */

/* @Service(name="TutorialBExample) */
/* ... operations ... */
```

9.6 Name Mapping

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

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

10 Packaging

10.1 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
              MyValue.componentType
              MyValueService.wsdl
       services/customer/
              CustomerService.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.c> 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 URI's. These URI's 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.

11 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.

11.1 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 *).
- DATAOBJECT. An SDO handle.

11.2 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.
- DATAOBJECT. An SDO handle. 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.

12 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 function

The following C keywords and constructs MUST NOT be used:

- Macros

2247 # 13 Conformance

A C Annotations

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

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

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

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.

- Function or Function Prototype

  The annotation immediately precedes the function definition or declaration.

  Example:

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

- Variable

  The annotation immediately precedes the variable definition.

  Example:

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

- Set of Functions Implementing a Service

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

  Example

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

- Set of Function Prototypes Defining an Interface

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

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

A.2 Interface Header Annotations

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

A.2.1 @Interface

Annotation that indicates the start of a new interface definition.

Corresponds to: interface.c element

Format:

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

where

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

Applies to: Set of functions defining an interface.

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

Example:

Implementation:

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

Service definition:

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

A.2.2 @Operation

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

Corresponds to: function element of an interface.c element

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

Format:

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

where

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

Applies to (library based implementations): Function declaration

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

Example:

Interface header (loans.h):

```
short internalFcn(char *param1, short param2);

/* @Operation(name=”getRate”) */
void rateFcn(char *cust, float *rate);
```

Interface definition:

```
<interface.c header="loans.h">
   <operation name=”getRate” />
</interface.c>
```

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

Format:

```
/* @Operation(name=”operationName”, input=”inputStuct”, output=”outputStruct”)
*/
```

where

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

Applies to (program based implementations): stuct declarations

Example

Interface header (loans.h):

```
/* @Operation(name=”getRate”, input=”rateInput”, output=”rateOutput”) */
struct rateInput {
    char cust[25];
    int  term;
};
struct rateOutput {
    float rate;
    int   rateClass;
};
```

Interface definition:

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

A.2.3 @Remotable

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

Corresponds to: remotable=”true” attribute of interface.c element.

Format:

```
/* @Remotable */
```

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

Applies to: Interface

Example

Interface header (LoanService.h):

```
/* @Remotable */
```

Service definition:

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

A.2.4 @Callback

Annotation on a service interface to specify the callback interface.

Corresponds to: callbackHeader attribute of interface.c element.

Format:

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

where

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

Applies to: Interface

Example

Interface header (MyService.h):

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

Service definition:

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

A.2.5 @OneWay

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

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

Format:

```
/* @OneWay */
```

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

Applies to: Function Prototype

Example

Interface header:

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

Service definition:

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

A.2.6 @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: Interface

Example

Interface header (LoanService.h):

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

Service definition:

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

A.2.7 @EndsConversation

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

Corresponds to: endsConversation=”true” attribute of function element of an interface.c element.

Format:

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

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

Applies to: Function Prototype

Example

Interface header:

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

Component definition:

```
<service name="LoanService">
   <interface.c header="LoanService.h">
      <operation name="cancelApplication" endsConversation="true" />
   </interface.c>
</service>
```

A.3 Implementation Annotations

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

A.3.1 @ComponentType

Annotation used to indicate the start of a new componentType.

Corresponds to: componentType attribute of implementation.c element.

Format:

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

Applies to: Set of services, references and properties

Example

Implementation:

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

Component definition:

```
<component name="LoanService">
   <implementation.c module="loan"
         componentType="LoanService" />
</component>
```

A.3.2 @Service

Annotation that indicates the start of a new service implementation.

Corresponds to: implementation.c element

Format:

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

where

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

Applies to: Set of functions implementing a service

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

Example

Implementation:

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

ComponentType definition:

```
<componentType name="LoanService">
   <service name="LoanService">
      <interface.c header="loans.h" />
   </service>
</componentType>
```

A.3.3 @Reference

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

Corresponds to: reference element of componentType element.

Format:

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

where

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

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

Applies to: Service

Example

Implementation:

```
/* @Reference(name="getRate", interfaceHeader="rates.h") */

/* @Reference(name="publishRate", interfaceHeader="myRates.h",
 *            required="false", multiple="yes"
 */
```

ComponentType definition:

```
<componentType name="LoanService">
   <reference name=”getRate”>
      <interface.c header=”rates.h”>
   </reference>
   <reference name=”publishRate” multiplicity=”0..n”>
      <interface.c header=”myRates.h”>
   </reference>
</componentType>
```

A.3.4 @Property

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

Corresponds to: property element of componentType element.

Format:

```
/* @Property(name=“propertyName”, type=”typeName”,
 *           default=”defaultValue”, required=”true”)
 */
```

where

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

Applies to: Variable

Example

Implementation:

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

ComponentType definition:

```
<componentType name="LoanService">
   <property name=”loanType” type=”xsd:int” />
</componentType>
```

A.3.5 @Scope

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

Corresponds to: scope attribute of implementation.c 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: Service

Example

Implementation:

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

Component definition:

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

A.3.6 @Init

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

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

Format:

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

Applies to: Function or Service

Example

Implementation:

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

A.3.7 @Destroy

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

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

Format:

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

Applies to: Function or Service

Example

Implementation:

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

A.3.8 @EagerInit

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

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

Format:

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

Applies to: Service

Example

Implementation:

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

Component definition:

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

A.3.9 @AllowsPassByReference

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

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

Format:

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

Applies to: Service or Function

Example

Implementation:

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

Component definition:

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

A.3.10 @ConversationAttributes

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

Corresponds to: conversationMaxAge, conversationMaxIdle or conversationSinglePrincipal attributes of implementation.c 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: Service

Example

Implementation:

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

Component definition:

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

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 function, to a C function declaration or to sets of functions implementing a service or sets of function declarations defining a service interface.

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

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

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

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

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

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

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

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

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

Format:

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

where

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

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

Examples:

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

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

A reference requiring support for confidentiality:

```
/* @Requires(CONFIDENTIALITY)
 * @Reference(interfaceHeader="SetBar.h") */
   void setBar(struct barType *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:

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

B.2 Specific Intent Annotations

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

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

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

```
/* @Integrity */
```

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

Format:

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

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

```
Intent ::= NCName
qualifiers ::= ”qualifier” | {“qualifier” [, “qualifier”] }
qualifier ::= NCName | NCName/qualifier
```

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

Example:

```
/* @Authentication( {“message”, “transport”} ) */
```

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

B.3 Application of Intent Annotations

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

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

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

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

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

B.4 Policy Annotation Scope

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

```
/* @Remotable
 * @Integrity(“transport”)
 * @Authentication
 * @Service(name=”HelloService”, interfaceHeader=”hellowservice.h” */

/* @Integrity
 * @Authentication(“message”)*/
   wchar_t* hello(wchar_t* message) {...}

/* @Integrity
 * @Authentication(“transport”) */
   wchar_t* helloThere() {...}
```

```
/* @Remotable
 * @Confidentiality(“message”)
 * @Service(name=”HelloHelperService”, interfaceHeader=”helloService.h” */

/* @Confidentiality(“transport”) */
   wchar_t* hello(wchar_t* message) {...}

/* @Authentication */
   wchar_t* helloWorld(){...}
```

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

- The effective intent annotation on the helloWorld function is @Authentication, and @Confidentiality(“message”).
- The effective intent annotation on the hello function of the HelloHelperService is @Confidentiality(“transport”),
- The effective intent annotation on the helloThere function of the HelloService is @Integrity and @Authentication(“transport”).
- The effective intent annotation on the hello function of the HelloService is @Integrity and @Authentication(“message”)

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

```
<?xml version="1.0" encoding="ASCII"?>

<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
                name="HelloServiceComposite" >
   <service name=”HelloService” requires=”integrity/transport
          authentication”>
          …
   </service>
   <service name=”HelloHelperService” requires=”integrity/transport
          authentication confidentiality/message”>
   …
   </service>
   ...

   <component name="HelloServiceComponent">*
          <implementation.c library="HelloService.dll"
                header=”HellowService.h”/>
          <operation name=”hello” requires=”integrity
                authentication/message”/>
          <operation name=”helloThere” requires=”integrity
                authentication/transport”/>
   </component>
   <component name="HelloHelperServiceComponent">*
          <implementation.c library="HelloService.dll"
                header=”HelloService.h” />
          <operation name=”hello” requires=”confidentiality/transport”/>
          <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 functions or function declarations 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.

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

B.6 Policy Set Annotations

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

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

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

Format:

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

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

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

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]**.

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 */
/* @Interface(name="HelloService") */
wchar_t* hello(wchar_t* helloMsg);

/* Implementation for ClientService */
#include "HelloService.h"

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

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

void clientMethod() {
   SCAREF serviceToken;
   int compCode, reason;
   wchar_t result[256];

   SCALocate(L"customerService", &serviceToken, &compCode, &reason);
   SCAInvoke(serviceToken, L"hello", sizeof(L"Hello World!"),
             L"Hello World!", sizeof(result), (void *)&result,
             &compCode, &reason);
}
```

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
 * @Interface(name=”AccountService”) */
struct AccountReport *getAccountReport(char *customerID);
```

The following is a full listing of the AccountService implementation, 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 “SCA.h”
#include “AccountService.h”;
#include “AccountDataService.h”;
#include “StockQuoteService.h”;

/* @RolesAllowed(“customers”)
 * @RunAs(“accountants” )
 * @Service(name=”AccountService”, interfaceHeader=”AccountService.h”) */

/* @Property */
   char currency[5] = "USD";

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

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

/* @RolesAllowed({“customers”, “accountants”}) */
struct AccountReport *getAccountReport(char *customerID) {
   struct AccountReport *accounts;
   struct CheckingAccount *checking;
   struct SavingsAccount *savings;
   struct StockAccount *stock;
   float balance;
   struct QuoteRequest stockToQuote;
   SCAREF, accountDataService, stockQuoteService;
   int compCode, Reason;
```

```
   accounts = (struct AccountReport *)malloc(sizeof(struct AccountReport));
   accounts->numAccounts = 0;

   SCALocate(L"accountDataService", &accountDataService, &compCode, &Reason);

   SCAInvoke(accountDataService, L"getCheckingAccount",
             sizeof(customerID), (void *)customerID,
             sizeof(struct CheckingAccount), (void *)checking,
             &compCode, &Reason);
   accounts->numAccounts += 1;
   strcpy(accounts->summary[0].number, checking->number);
   strcpy(accounts->summary[0].type, "checking");
   accounts->summary[0].balance = fromUSDollarToCurrency(checking->balance);

   SCAInvoke(accountDataService, L"getSavingsAccount",
             sizeof(customerID), (void *)customerID,
             sizeof(struct SavingsAccount), (void *)savings,
             &compCode, &Reason);
   accounts->numAccounts += 1;
   strcpy(accounts->summary[1].number, savings->number);
   strcpy(accounts->summary[1].type, "savings");
   accounts->summary[1].balance = fromUSDollarToCurrency(savings->balance);

   SCAInvoke(accountDataService, L"getStockAccount",
             sizeof(customerID), (void *)customerID,
             sizeof(struct StockAccount), (void *)stock,
             &compCode, &Reason);
   accounts->numAccounts += 1;
   strcpy(accounts->summary[2].number, stock->number);
   strcpy(accounts->summary[2].type, "stock");
   strcpy(stockToQuote.symbol, stock->symbol);
   stockToQuote.quantity = stock->quantity;
   SCALocate(L"stockQuoteService", &stockQuoteService, &compCode, &Reason);
   SCAInvoke(stockQuoteService, L"getQuote",
             sizeof(struct QuoteRequest), (void *)&stocktoQuote,
             sizeof(balance), (void *)&balance,
             &compCode, &Reason);
   accounts->summary[2].balance = fromUSDollarToCurrency(balance);

   return accounts;
}

/* @PermitAll */
float fromUSDollarToCurrency(float value){

   if (!strcmp(currency, "USD"))
      return value;
   else if (!(strcmp(currency, "EURO"))
      return value * 0.8;
   else
      return 0.0;
}
```

In this example, the implementation 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(..) funcgtion is marked with @RolesAllowed({"customers", "accountants"}), which indicates that this function can be called by both customers and accountants.

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

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-c-1.1-schema.xsd

and

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

The following copies are provided for reference.

C.1 sca-interface-c-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"
      elementFormDefault="qualified">

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

   <element name="interface.c" type="sca:CInterface"
         substitutionGroup="sca:interface"/>

   <complexType name="CInterface">
      <complexContent>
         <extension base="sca:Interface">
            <sequence>
               <element name="function" type="sca:CFunction"
                     minOccurs="0" maxOccurs="unbounded" />
               <element name="callbackFunction" type="sca:CFunction"
                     minOccurs="0" maxOccurs="unbounded" />
               <any namespace="##other" processContents="lax"
                     minOccurs="0" maxOccurs="unbounded"/>
            </sequence>
            <attribute name="header" type="Name" use="required"/>
            <attribute name="callbackHeader" type="Name" use="optional"/>
            <attribute name="remotable" type="boolean" use="optional"/>
            <anyAttribute namespace="##other" processContents="lax"/>
         </extension>
      </complexContent>
   </complexType>

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

</schema>
```

C.2 sca-implementation-c-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"
      elementFormDefault="qualified">

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

   <element name="implementation.c" type="sca:CImplementation"
         substitutionGroup="sca:implementation" />

   <complexType name="CImplementation">
      <complexContent>
         <extension base="sca:Implementation">
            <sequence>
               <element name="operation" type="sca:CImplementationFunction"
                     minOccurs="0" maxOccurs="unbounded" />
               <any namespace="##other" processContents="lax"
                     minOccurs="0" maxOccurs="unbounded"/>
            </sequence>
            <attribute name="module" type="NCName" use="required"/>
            <attribute name="location" type="Name" use="optional"/>
            <attribute name="library" type="boolean" use="optional"/>
            <attribute name="componentType" type="Name"/>
            <attribute name="scope" type="sca:CImplementationScope"
                  use="optional"/>
            <attribute name="eagerInit" type="boolean" use="optional"/>
            <attribute name="init" type="boolean" use="optional"/>
            <attribute name="destoy" type="boolean" use="optional"/>
            <attribute name="allowsPassByReference" type="boolean"
                  use="optional"/>
            <attribute name="conversationMaxAge" type="string"
                  use="optional"/>
            <attribute name="conversationMaxIdle" type="string"
                  use="optional"/>
            <attribute name="conversationSinglePrincipal" type="boolean"
                  use="optional"/>
            <anyAttribute namespace="##other" processContents="lax"/>
         </extension>
      </complexContent>
   </complexType>

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

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

</schema>
```

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 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.

E Acknowledgements

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

Participants:

Andrew Borley, IBM
Bryan Aupperle, IBM
David Haney, Rogue Wave Software
Mike Edwards, IBM
Pete Robbins, IBM

3261 # F Non-Normative Text

G Revision History

[optional; should not be included in OASIS Standards]

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-20, CCPP-27 and equivalent work for CCPP-31
2	2007-11-08	Bryan Aupperle	• Update MUST/must, SHOULD/should and MAY/may to reflect TC use or RFC 2119 • Incoroporate changes for CCPP-7, CCPP-8, CCPP-11, CCPP-16 and CCPP-18
1	2007-09-28	Bryan Aupperle	• Initial conversion of the OSOA 1.0 specification to the OASIS template.