SCA Service Component Architecture

Assembly Model Specification

SCA Version 0.9, November 2005

Technical Contacts:	Michael Beisiegel	IBM Corporation
	Henning Blohm	SAP AG
	Dave Booz	IBM Corporation
	Jean-Jacques Dubray	SAP AG
	Mike Edwards	IBM Corporation
	Bill Flood	Sybase, Inc.
	Bruce Ge	Siebel Systems, Inc.
	Oisin Hurley	IONA Technologies plc.
	Dan Kearns	Siebel Systems, Inc.
	Mike Lehmann	Oracle Corporation
	Jim Marino	BEA Systems, Inc.
	Martin Nally	IBM Corporation
	Greg Pavlik	Oracle Corporation
	Michael Rowley	BEA Systems, Inc.
	Adi Sakala	IONA Technologies plc.
	Chris Sharp	IBM Corporation
	Ken Tam	BEA Systems, Inc

Copyright Notice

License

The Service Component Architecture Specification is being provided by the copyright holders under the following license. By using and/or copying this work, you agree that you have read, understood and will comply with the following terms and conditions:

Permission to copy and display the Service Component Architecture Specification and/or portions thereof, without modification, in any medium without fee or royalty is hereby granted, provided that you include the following on ALL copies of the Service Component Architecture Specification, or portions thereof, that you make:

1. A link or URL to the Service Component Architecture Specification at these locations:
   - http://dev2dev.bea.com/technologies/commonj/index.jsp
   - http://www.ibm.com/developerworks/library/specification/ws-sca/
   - http://www.iona.com/devcenter/sca/
   - http://oracle.com/technology/webservices/sca
   - https://www.sdn.sap.com/
   - http://www.sybase.com

2. The full text of this copyright notice as shown in the Service Component Architecture Specification.

BEA, IBM, IONA, Oracle, SAP, Siebel, Sybase (collectively, the “Authors”) agree to grant you a royalty-free license, under reasonable, non-discriminatory terms and conditions to patents that they deem necessary to implement the Service Component Architecture Specification.

THE Service Component Architecture SPECIFICATION IS PROVIDED "AS IS," AND THE AUTHORS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, REGARDING THIS SPECIFICATION AND THE IMPLEMENTATION OF ITS CONTENTS, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT OR TITLE.

THE AUTHORS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATING TO ANY USE OR DISTRIBUTION OF THE SERVICE DATA OBJECTS SPECIFICATION.

Status of this Document

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

Table of Contents

1 Assembly Model

1.1 Introduction

This chapter describes the ***SCA Assembly Model, which*** covers

- A model for the assembly of tightly coupled services
- A model for the assembly of loosely coupled service-oriented systems

The chapter starts with a short overview of the SCA Assembly Model.

The second part of this chapter describes the design-time Assembly of tightly coupled services through the use of Modules.

The third part of this chapter covers the SCA deployment-time artifacts - how an ***SCA system*** (i.e. the SCA runtime) composes loosely coupled services using Subsystems which configure modules.

The next part of the chapter covers Bindings, which describes how SCA makes use of specific access methods such as Web services.

The final part of this chapter defines how the SCA assembly model can be extended.

1.2 Overview

The SCA Assembly Model consists of a series of artifacts, which are defined by elements contained in XML files. An SCA runtime may have other non-standard representations of the artifacts represented by these XML files, and may allow for the configuration of systems to be modified dynamically. However, the XML files define the portable representation of the SCA artifacts.

The basic artifact is the Module, which is the unit of deployment for SCA and which holds Services which can be accessed remotely. A module contains one or more Components, which contain the business function provided by the module. Components offer their function as services, which can either be used by other components within the same module or which can be made available for use outside the module through Entry Points. Components may also depend on services provided by other components – these dependencies are called References. References can either be linked to services provided by other components in the same module, or references can be linked to services provided outside the module, which can be provided by other modules. References to services provided outside the module, including services provided by other modules, are defined by External Services in the module. Also contained in the module are the linkages between references and services, represented by Wires.

A Component consists of a configured Implementation, where an implementation is the piece of program code implementing business functions. The component configures the implementation with specific values for settable Properties declared by the implementation. The component can also configure the implementation with wiring of references declared by the implementation to specific target services.

Modules are deployed within an SCA System. An SCA System represents a set of services providing an area of business functionality that is controlled by a single organization. As an example, for the accounts department in a business, the SCA System might cover all financial

related function, and it might contain a series of modules dealing with specific areas of accounting, with one for customer accounts, another dealing with accounts payable. To help build and configure the SCA System, Subsystems are used to group and configure related modules. Subsystems contain Module Components, which are configured instances of modules. Subsystems, like modules, also have Entry Points and External Services which declare external services and references which exist outside the system. Subsystems can also contain Wires which connect together the module components, entry points and external services.

1.2.1 Diagrams used to Represent SCA Artifacts

This document introduces diagrams to represent the various SCA artifacts, as a way of visualizing the relationships between the artifacts in a particular assembly. These diagrams are used in this document to accompany and illuminate the examples of SCA artifacts.

The following picture illustrates some of the features of a module assembly diagram:

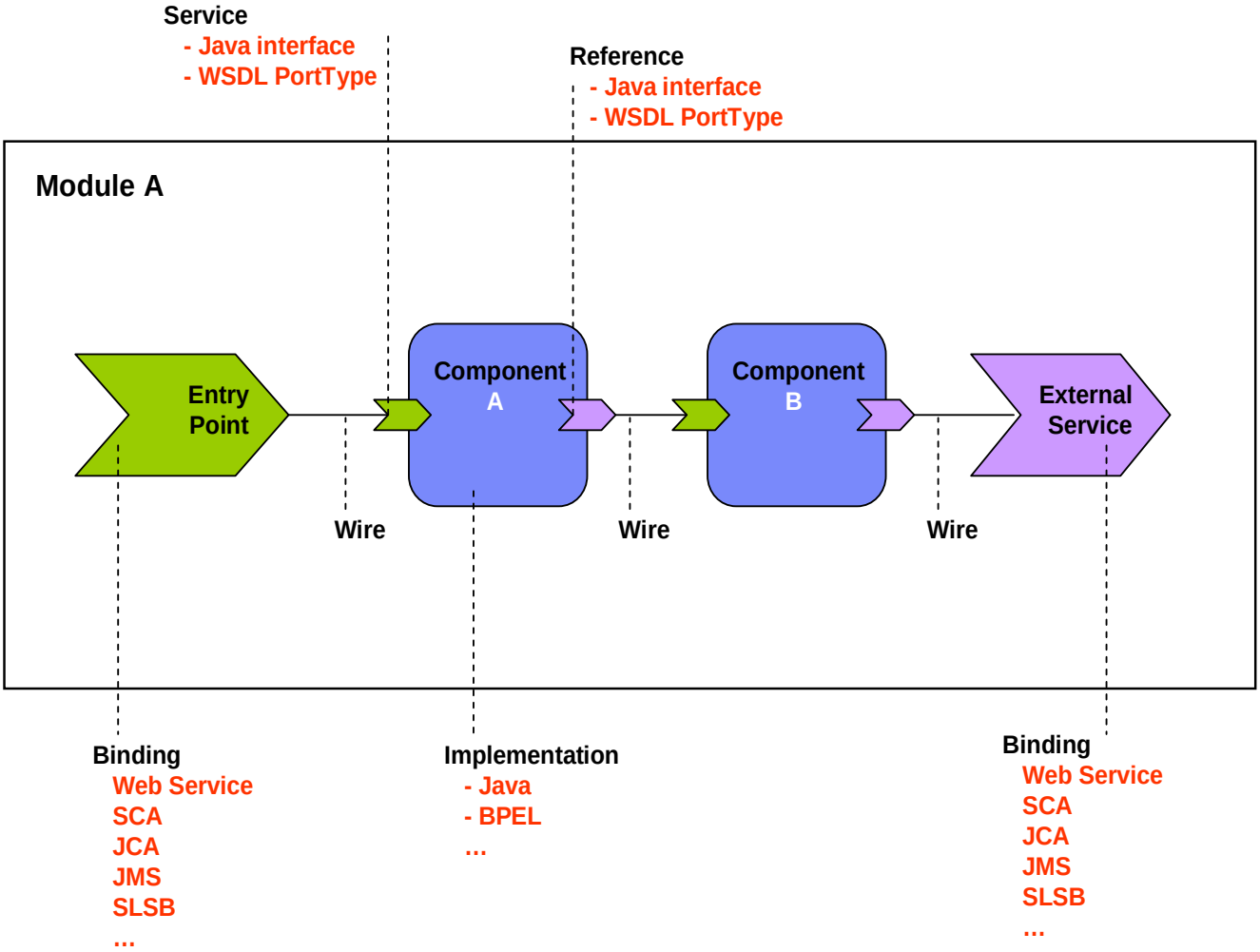


Figure 1: Module Assembly Diagram

The following picture illustrates some of the features of a system assembly diagram:

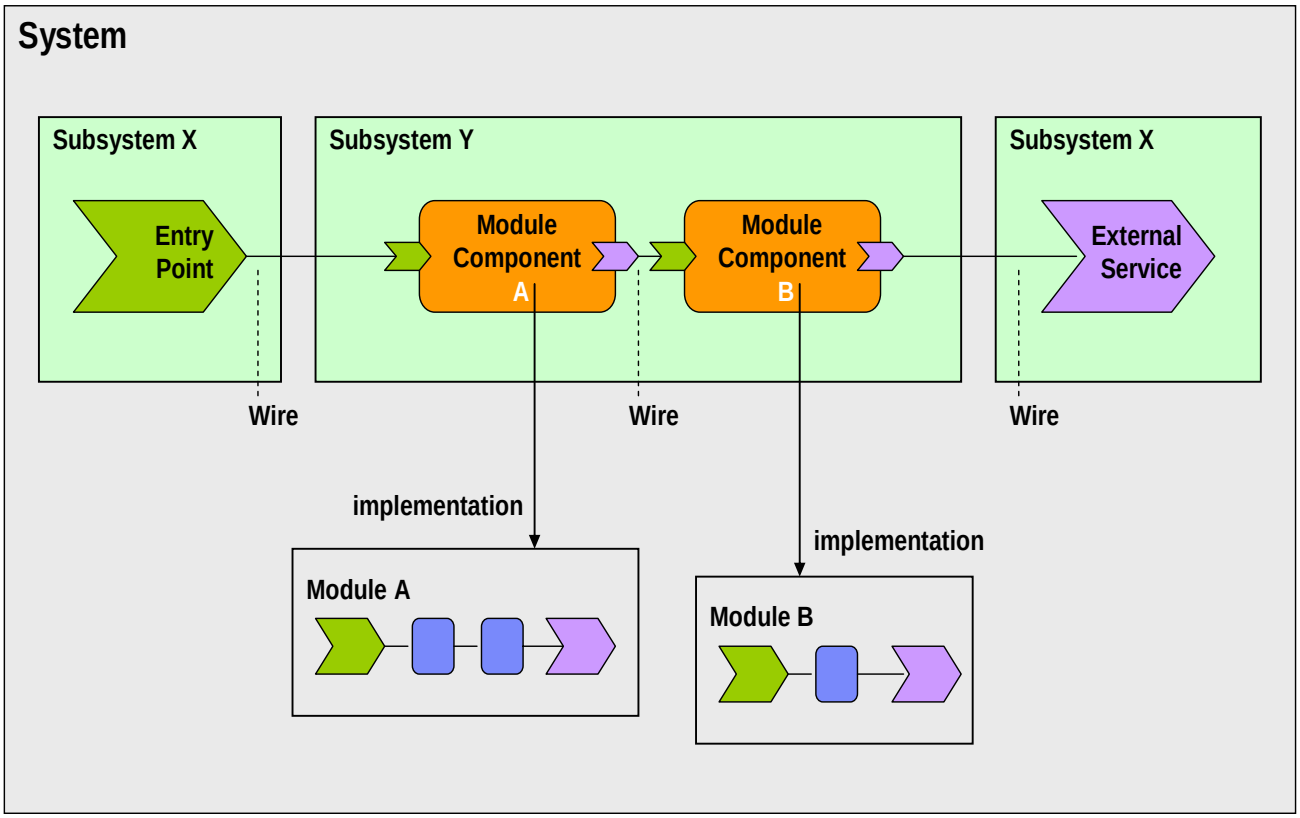


Figure 2: System Assembly Diagram

1.3 Module

An SCA module is the largest composition of tightly-coupled components that are developed and deployed together. It is the basic unit of loosely-coupled composition within an SCA System. An ***SCA Module*** contains a set of Components, External Services, Entry Points, and the Wires that interconnect them. Modules contribute service implementations to an ***SCA System***.

A ***module*** has the following normative characteristics:

- It defines a boundary for Component visibility. Components may not be directly referenced outside of a module.
- Within a module, service invocations are performed as local operations with by-reference semantics, except for interfaces declared as remotable. Between modules, service invocations are performed remotely using by-value semantics.
- It defines a unit of deployment. Modules are used to contribute business services to an SCA system.

Modules are defined by a ***module element*** in an ***sca.module*** file.

The following shows the module schema.

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

   name="xs:NCName" >

   <entryPoint name="xs:NCName" multiplicity="0..1 or 1..1 or 0..n or 1..n"?>*
         <interface.interface-type/>
         <binding.binding-type uri="xs:anyURI"/>+
         <reference>wire-target-URI</reference>
   </entryPoint>

   <component name="xs:NCName">*
         <implementation.implementation-type/>
         <properties>?
               <v:property-name>property-value</v:property-name>+
         </properties>
         <references>?
               <v:reference-name>wire-target-URI</v:reference-name>+
         </references>
   </component>

   <externalService name="xs:NCName">*
         <interface.interface-type/>+
         <binding.binding-type uri="xs:anyURI"/>*
   </externalService>

   <wire>*
         <source.uri>wire-source-URI</source.uri>
         <target.uri>wire-target-URI</target.uri>
   </wire>

</module>
```

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

- ***name*** – the name of the module, has to be unique in an SCA system. The form of a module name is also restricted so that it must not contain the characters "/" or "#" since these are used as separators between the elements of a fully qualified name at the system level.

Note: The Eclipse naming convention for plugins provides a good way to achieve unique names, e.g. com.mycompany.myvaluemodule . This format is recommended but is not normative.

Modules contain ***zero or more entry points***, ***components***, ***external services***, and ***wires***. These artifacts are described in detail in the following sections. Components contain the business logic of the Module. Wires describe the Service connections to and from components. Entry Points define the public services provided by the Module, which can be accessed from outside the module. External servicess represent dependencies that the Module has on services provided elsewhere, outside the module.

1.3.1 Component

Components are configured ***instances*** of ***implementations.*** Components provide and consume services. More than one component can use and configure the same implementation, where each component configures the implementation differenty. For example each component may configure a reference of the same implementation to consume a different service.

An implementation defines the aspects configurable by a component in the form of a ***component type.*** SCA allows for many different implementation technologies such as Java™, Business Process Execution Language [7] (BPEL), C++. SCA defines an ***extensibility mechanism*** for introducing new implementation types. The current specification does not mandate the implementation technologies to be supported by an SCA run-time, vendors may choose to support the ones that are important for them.

Components are defined in an ***sca.module*** file. A component is represented by a ***component element*** which is a child of the module element. There can be ***zero or more*** component elements within a module. The following snippet shows the module schema with the schema for the component child element.

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

   name="xs:NCName" >

   …

   <component name="xs:NCName">*
         <implementation.implementation-type/>
         <properties>?
               <v:property-name override="no or may or must"?
                     modulePropertyName="xs:NCName"?>
                     property-value
               </v:property-name>+
         </properties>
         <references>?
               <v:reference-name>wire-target-URI</v:reference-name>+
         </references>
   </component>

   …
```

```
</module>
```

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

- ***name*** – the name of the component. The name must be:
  - unique across all the components in the module
  - no entry point in the same module can have the same name as a component since they both can be sources of wires
  - no external service in the same module can have the same name as a component since they both can be targets of wires

A component element has one ***implementation element*** as its child, which points to the implementation. The name of the implementation element is architected; it is in itself a qualified name. The first qualifier is always named implementation, and the second qualifier names the respective implementation-type (e.g. implementation.java, implementation.bpel for Java and BPEL implementation types respectively). The following two snippets show implementation elements for the Java and BPEL implementation types:

```
<implementation.java class="services.myvalue.MyValueServiceImpl"/>
```

```
<implementation.bpel process="…"/>
```

Before describing the configuration aspects of components (defined by the child elements Properties and References which are described in the section “Component Configuration”), component types and implementations are described, since it is aspects of the component type that are configured by components – and the configurable aspects are defined by the component type.

1.3.2 Component Type

Component types represent configurable aspects of an implementation. A component type consists of services that are offered, references to other services that can be set and properties that can be set. The settable properties and the settable references to services are configured by a component which uses the implementation.

The ***Component type is calculated in two steps*** where the second step adds to the information found in the first step. Step one is introspecting the implementation (if possible), including the inspection of implementation annotations (if available). Step two looks for an SCA component type file. Component type information found in the component type file must be compatible with the equivalent information found from inspection of the implementation. The component type file can specify partial information, with the remainder being derived from the implementation. This process allows for the possibility of including all the component type information inside the implementation, for example as code annotations, while also providing a mechanism for implementation types where annotations of this kind are not possible.

The ***component type file*** is named like the implementation file but has the extension “***.componentType***”. The component type is defined by an ***componentType element*** in the file. The location of the component type file depends on the language used for the component

implementation: it is described in the respective client and implementation model specification for that language.

The componentType element can contain Service elements, Reference elements and Property elements.

The following snippet shows the componentType schema.

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

   <service name="xs:NCName">*
         <interface.interface-type/>
   </service>

   <reference name="xs:NCName" multiplicity="0..1 or 1..1 or 0..n or 1..n"?>*
         <interface.interface-type/>
   </reference>

   <property name="xs:NCName" type="xs:QName" many="xs:boolean"?
               default="xs:string"? required="xs:boolean"?/>*

</componentType>
```

A Service represents an addressable interface on a component type. The service is represented by a ***service element*** which is a child of the componentType element. There can be ***zero or more*** service elements in a componentType.

The service element has the following ***attribute***:

- ***name*** – the name of the service. The name must be unique across all the services of an implementation

A service has ***1 interface***. The interface is described by an ***interface element*** which is a child element of the service element. For details on the interface element see the interface section.

A service can be ***remotable***. Whether a service is remotable is defined by its interface. Remotable services are services that ***can be made available*** through entry points. Services made available in this way can be accessed by clients outside of the module containing the component that provides the service. For details on remotable interfaces see the interface section.

A ***Reference*** represents a requirement that the component type has on an interface to be provided by another service. The reference is represented by a ***reference element*** which is a child of the componentType element. There can be ***zero or more*** reference elements in an component type definition.

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

- ***name*** – the name of the reference, the name must be unique across all the references of an implementation
- ***multiplicity (optional)*** - Defines the number of wires that can connect the reference to target services. The multiplicity can have the following values
    - 1..1 (default) – one wire can have the reference as a source

- 0..1 – zero or one wire can have the reference as a source
- 1..n – one or more wires can have the reference as a source
- 0..n - zero or more wires can have the reference as a source

A reference has ***1 interface***. The interface is described by an ***interface element*** which is a child element of the reference element. For details on the interface element see the interface section.

A reference can be ***remote***. Whether a reference is remote is defined by its interface. If the interface is defined as remotable then the reference is remote. The implementation using the reference will experience remote semantics when interacting with the remotable service bound to the remote reference. For details on remotable interfaces see the interface section.

Multiplicity defines the number of services that can be wired to the reference:

- 1..1 is the simplest and most common case, where the implementation expects exactly one service to be wired. The implementation will fail at runtime if no service is wired to the reference. A warning will be generated by the SCA runtime at deployment time if this is the case.
- 0..1 caters for an implementation that can tolerate the case where no service is wired to the reference, which might be the case where the service being invoked is a logging service, for example, where correct operation of the component will occur even if no logging service is wired.
- 0..n and 1..n cater for an implementation that is prepared to handle multiple target services. The two cases differ according as to whether the implementation can tolerate no service being wired to the reference – 0..n allows for no service to be wired while 1..n requires at least one. In both these cases, the implementation receives a ***collection*** of service references rather than a single one. It is up to the implementation to invoke each service in turn, as required by its business function.

  0..n is likely to find use in "publish/subscribe" event type relationships, where many services can receive a message from an implementation, but where the implementation itself can operate successfully even if there are none. 1..n might be used in situations where the implementation wants to invoke the same service interface against many services, perhaps to compare the results – an example might be doing a price comparison for widgets from multiple suppliers. In this example, each wire represents a connection to the widget price query service of a different supplier.

Properties allow for the configuration of a component type with externally set values. Each Property is defined as a property element. The componentType element can have zero or more property elements as its children.

The property element has the following attributes:

- ***name*** – (required) the name of the property
- ***type*** – (required) the type of the property - the qualified name of an XML schema simple type
- ***many*** – (optional) whether the property is single-valued (false) or multi-valued (true). The default is ***false***. In the case of a multi-valued property, it is presented to the implementation as a Collection of property values.

- ***default*** – (optional) default value for the property
- ***required*** – (optional) If true, then the module file must either provide a value for the property, or it must declare the property with an override attribute of must so that the property value is set at the subsystem level. If required is true then a default value must not be specified. If required is false, no value need be supplied for the property at either module or subsystem level. Default is ***false***.

Note: The assembly model may support complex property types in the future.

1.3.2.1 Example ComponentType

The following snippet shows the contents of the componentType file for the MyValueServiceImpl implementation. The componentType file shows the services, references, and properties of the MyValueServiceImpl implementation. In this case, Java is used to define interfaces and property values:

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

   <service name="MyValueService">
         <interface.java interface="services.myvalue.MyValueService"/>
   </service>

   <reference name="customerService">
         <interface.java interface="services.customer.CustomerService"/>
   </reference>
   <reference name="stockQuoteService">
         <interface.java interface="services.stockquote.StockQuoteService"/>
   </reference>

   <property name="currency" type="xsd:string" default="USD"/>

</componentType>
```

1.3.3 Implementation

Component***Implementations*** are concrete implementations of business function which provide services and/or consume services. SCA allows you to choose from any one of a wide range of ***implementation technologies***, such as Java, BPEL or C++.

Services, references and properties are the configurable aspects of an implementation, SCA refers to them collectively as the ***component type***.

At runtime, an ***implementation instance*** is a specific runtime instantiation of the implementation – its specific form depends on the implementation technology used. The implementation instance derives its business logic from the implementation on which it is based, but the values for its properties and references are derived from the component which uses the implementation. The overall relationship between component types, components, implementations and implementation instances are described in the following diagram:

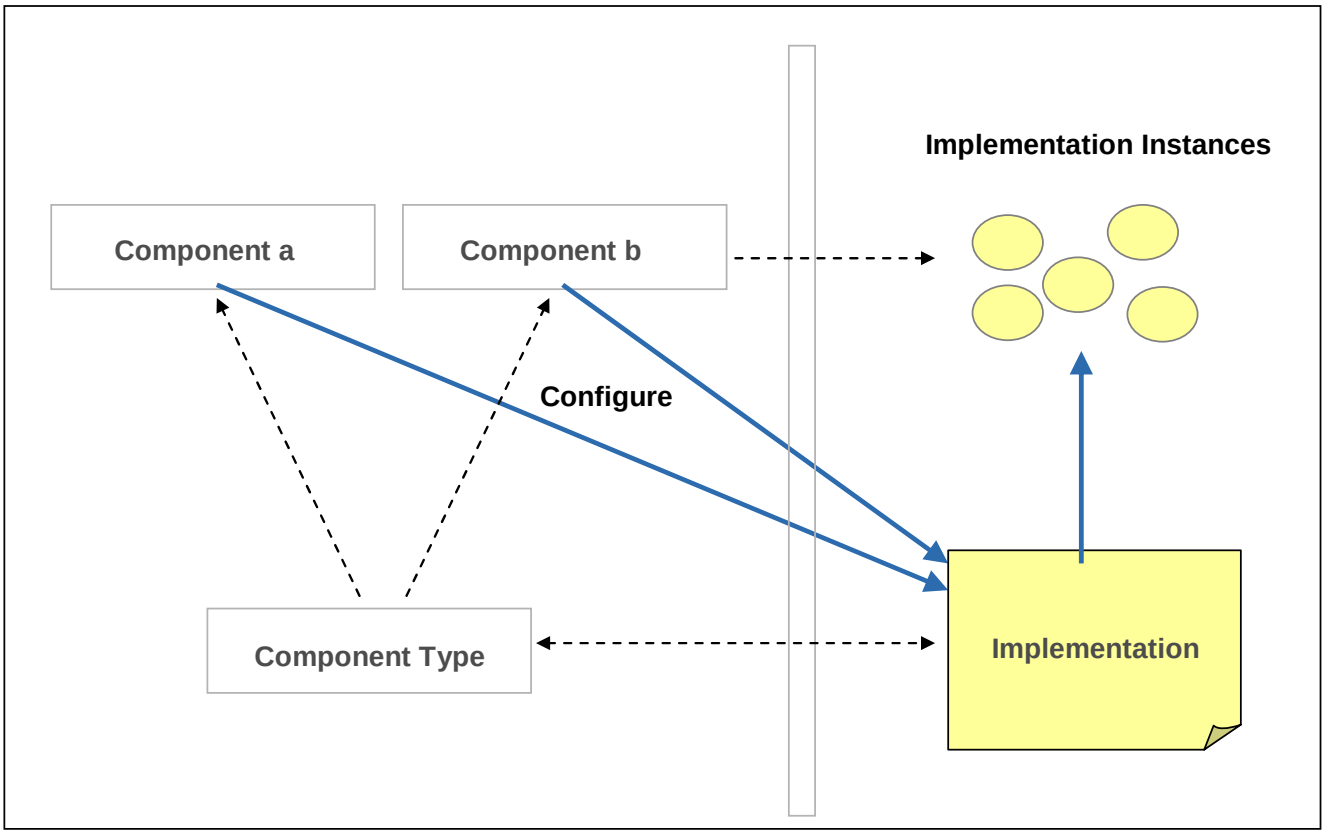


Figure 3: Relationship of Component and Implementation

1.3.3.1 Example Implementation

The following is an example implementation, written in Java, taken from the SCA Example Code document [3]. This is the ***AccountServiceImpl***.

AccountServiceImpl implements the ***AccountService*** interface, which is defined via a Java interface:

```
package services.account;

@Remotable
public interface AccountService{

    public AccountReport getAccountReport(String customerID);
}
```

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

```
package services.account;

import java.util.List;

import commonj.sdo.DataFactory;
```

```
import org.osoa.sca.annotations.Property;
import org.osoa.sca.annotations.Reference;

import services.accountdata.AccountDataService;
import services.accountdata.CheckingAccount;
import services.accountdata.SavingsAccount;
import services.accountdata.StockAccount;
import services.stockquote.StockQuoteService;

public class AccountServiceImpl implements AccountService {

    @Property
    private String currency = "USD";

    @Reference
    private AccountDataService accountDataService;
    @Reference
    private StockQuoteService stockQuoteService;

    public AccountReport getAccountReport(String customerID) {

     DataFactory dataFactory = DataFactory.INSTANCE;
     AccountReport accountReport = (AccountReport)dataFactory.create(AccountReport.class);
     List accountSummaries = accountReport.getAccountSummaries();

     CheckingAccount checkingAccount = accountDataService.getCheckingAccount(customerID);
     AccountSummary checkingAccountSummary = (AccountSummary)dataFactory.create(AccountSummary.class);
     checkingAccountSummary.setAccountNumber(checkingAccount.getAccountNumber());
     checkingAccountSummary.setAccountType("checking");
     checkingAccountSummary.setBalance(fromUSDollarToCurrency(checkingAccount.getBalance()));
     accountSummaries.add(checkingAccountSummary);

     SavingsAccount savingsAccount = accountDataService.getSavingsAccount(customerID);
     AccountSummary savingsAccountSummary = (AccountSummary)dataFactory.create(AccountSummary.class);
     savingsAccountSummary.setAccountNumber(savingsAccount.getAccountNumber());
     savingsAccountSummary.setAccountType("savings");
     savingsAccountSummary.setBalance(fromUSDollarToCurrency(savingsAccount.getBalance()));
     accountSummaries.add(savingsAccountSummary);

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

     return accountReport;
    }

    private float fromUSDollarToCurrency(float value){

     if (currency.equals("USD")) return value; else
     if (currency.equals("EURO")) return value * 0.8f; else
     return 0.0f;
    }
}
```

This sample uses Java annotations to mark the various SCA aspects of the code. These include the @Property and @Reference tags. To make the SCA aspects of the implementation clearer, the following is the equivalent SCA componentType definition for the AccountServiceImpl:

```
<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/0.9"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <service name="AccountService">
         <interface.java interface="services.account.AccountService"/>
```

```
    </service>
    <reference name="accountDataService">
          <interface.java interface="services.accountdata.AccountDataService"/>
    </reference>
    <reference name="stockQuoteService">
          <interface.java interface="services.stockquote.StockQuoteService"/>
    </reference>

    <property name="currency" type="xsd:string" default="USD"/>

</componentType>
```

For full details about implementations, look at the Client and Implementation Specification and the SCA Example Code document.

1.3.4 Interface

Interfaces define one or more business functions. These business functions are provided by Services and are used by References. Services are defined by the Interface which they implement. SCA currently supports the following interface type systems:

- Java interfaces
- WSDL 1.1 portTypes
- WSDL 2.0 interfaces

(WSDL: Web Services Definition Language [8])

The following snippet shows the schema for the Java interface element.

```
<interface.java interface="NCName" … />
```

The interface.java element has the following attributes:

- ***interface*** – the fully qualified name of the Java interface

The following sample shows a sample for the Java interface element.

```
<interface.java interface="services.stockquote.StockQuoteService"/>
```

The following snippet shows a sample for the WSDL portType (WSDL 1.1) or WSDL interface (WSDL 2.0) element.

```
<interface.wsdl interface="xs:anyURI" … />
```

The interface.wsdl element has the following attributes:

- ***interface*** – URI of the portType/interface with the following format
    - <WSDL-namespace-URI>#wsdl.interface(<portTypeOrInterface-name>)

The following snippet shows a sample for the WSDL portType/interface element.

```
<interface.wsdl interface="http://www.stockquote.org/StockQuoteService#
                                                wsdl.interface(StockQuote)"/>
```

For the Java interface type system, ***arguments and return*** of the service methods are described using Java classes or simple Java types. SDO-generated Java classes are the preferred form of Java class because of their integration with XML technologies.

For WSDL 1.1, the interface attribute points to a portType in the WSDL. For WSDL 2.0, the interface attribute points to an interface in the WSDL. For the WSDL 1.1 portType and WSDL 2.0 interface type systems, arguments and return of the service methods are described using XML schema. Arguments described in XML Schema are exposed to programmers as SDO DataObjects.

1.3.4.1 Local and Remotable Interfaces

Whether a service of a component implementation is remotable is defined by the interface of the service. In the case of Java this is defined by adding the ***@Remotable*** annotation to the Java interface (see Client and Implementation Model Specification for Java). WSDL defined interfaces are always remotable.

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

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

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

Here is a snippet which shows an example of a remotable java interface:

```
package services.hello;

@Remotable
public interface HelloService {

   String hello(String message);
}
```

Services provided by an external service are always remotable.

It is possible for the implementation of a remotable service to indicate that it can be called using by-reference data exchange semantics when it is called from a component in the same module. This can be used to improve performance for service invocations within a module. This can be done using the @AllowsPassByReference annotation (see the Java Client and Implementation Specification).

A service typed by a local interface can only be called by clients that are part of the same module as the component that implements the local service. Local services cannot be published through entry points. In the case of Java a local service is defined by a Java interface definition without a ***@Remotable*** annotation.

The style of local interfaces is typically ***fine grained*** and intended for ***tightly coupled*** interactions. Local service interfaces can make use of ***method overloading***.

The data exchange semantic for calls to services typed by local interfaces is ***by-reference***.

1.3.4.2 Bidirectional Interfaces

The relationship of a business service to another business service is often peer-to-peer, requiring a two-way dependency at the service level. In other words, a business service represents both a consumer of a service provided by a partner business service and a provider of a service to the partner business service. This is especially the case when the interactions are based on asynchronous messaging rather than on remote procedure calls. The notion of bidirectional interfaces is used to directly model peer-to-peer bidirectional business service relationships.

An interface element for a particular interface type system must allow the specification of an optional callback interface. If a callback interface is specified, the interface element characterizes a conversational relationship, and SCA refers to it as a bidirectional interface.

The following snippet shows the interface element for Java with an optional callbackInterface attribute.

```
<interface.java      interface="services.invoicing.ComputePrice"
                     callbackInterface="services.invoicing.InvoiceCallback"/>
```

If a service is defined using a bidirectional interface element then its implementation implements the interface, and its implementation uses the callback interface to converse with the client that called the service interface.

If a reference is defined using a bidirectional interface element, the client component implementation using the reference calls the referenced service using the interface. The client component implementation must implement the callback interface.

Callbacks may be used for both remotable and local services. Either both interfaces of a bidirectional service must be remotable, or both must be local. It is illegal to mix the two.

Facilities are provided within SCA which allow a different component to provide a callback interface than the component which was the client to an original service invocation. How this is done can be seen in the SCA Java Client and Implementation Specification (section named “Passing Conversational Services as Parameters”).

1.3.4.3 Mapping of Java Interfaces to & from WSDL Interfaces

A Wire can connect between interfaces defined by different languages ie. Java interfaces and WSDL portTypes/WSDL interfaces in either direction, as long as the operations defined by the

two interface types are equivalent. They are equivalent if the operation(s), parameter(s), return value(s) and faults/exceptions map to each other.

SCA maps Java interfaces to and from WSDL interfaces using the mapping defined by the JAX-WS specification [4], with the following differences:

- Complex parameters and return values in WSDL operations may be mapped as SDOs, using the mapping defined in the SDO 2.0 specification [2]
- Holder classes are not supported

For the sake of determining the equivalence of a Java interface and a WSDL interface, the extra asynchronous methods that are implied by the JAX-WS mapping are not considered.

1.3.5 Component Configuration

Components are configured instances of ***componentTypes***, and in particular can apply specific values to properties declared by the componentType and can wire references declared by the componentType to specific targets.

Repeating the module schema snippet which shows the schema for the component element:

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

   name="xs:NCName" >

   …

   <component name="xs:NCName">*
         <implementation.implementation-type/>
         <properties>?
               <v:property-name  override="no or may or must"?
                                 modulePropertyName="xs:NCName"?>
                     property-value
               </v:property-name>+
         </properties>
         <references>?
               <v:reference-name>wire-target-URI</v:reference-name>+
         </references>
   </component>

   …

</module>
```

The component element can have ***zero or one properties element*** as its child, used to configure data values on the implementation. The properties element has one or more ***<v:property-name> elements*** as its children. Each property-name element provides a value which is passed to the implementation. The properties that can be configured are defined by the implementation. Where a property is declared as multi-valued, multiple <v:property-name> elements can be present for one property-name.

Note: The property name elements have to use the ***v*** namespace prefix to indicate that they are user defined through the component implementation, and not defined by the SCA XML schema.

Each property element may also optionally specify an ***override*** attribute, whose value is one of "no", "may" or "must", which defines whether the property value can be overridden at the subsystem level. For more information about this see the section on Overrides.

The component element can have ***zero or one references element*** as its child. The references element has one or more **<v:reference-name> elements** as its children. The references that can be configured are defined by the componentType. A given reference-name element can occur more than once if the multiplicity for the reference is 0..n or 1..n. The value of the <reference-name> element ***wires*** the reference to a service that resolves the reference. For more details on reference wiring see the section on Wires.

Note: The reference name elements have to use the ***v*** namespace prefix to indicate that they are user defined through the component implementation, and not defined by the SCA XML schema.

1.3.5.1 Example Component

The following figure shows the ***component symbol*** that we use to represent a component in a module diagram.

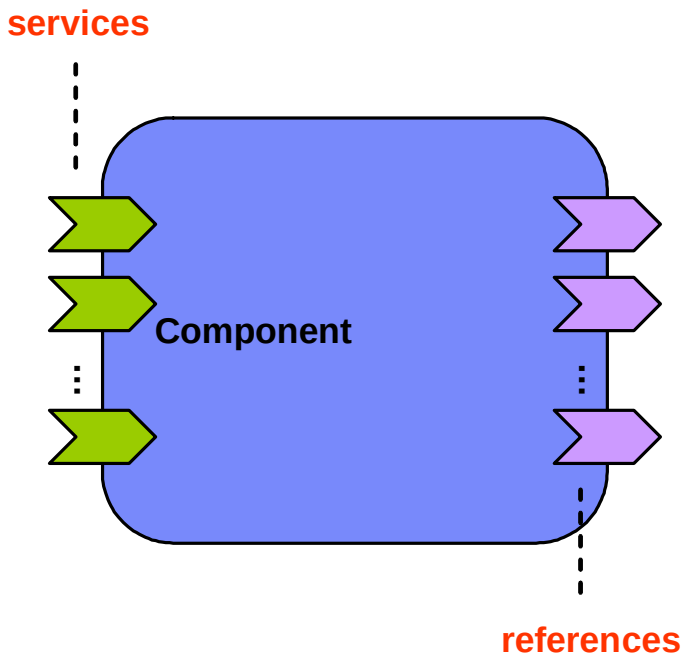


Figure 4: Component symbol

The following figure shows the module diagram for the MyValueModule containing the MyValueServiceComponent.

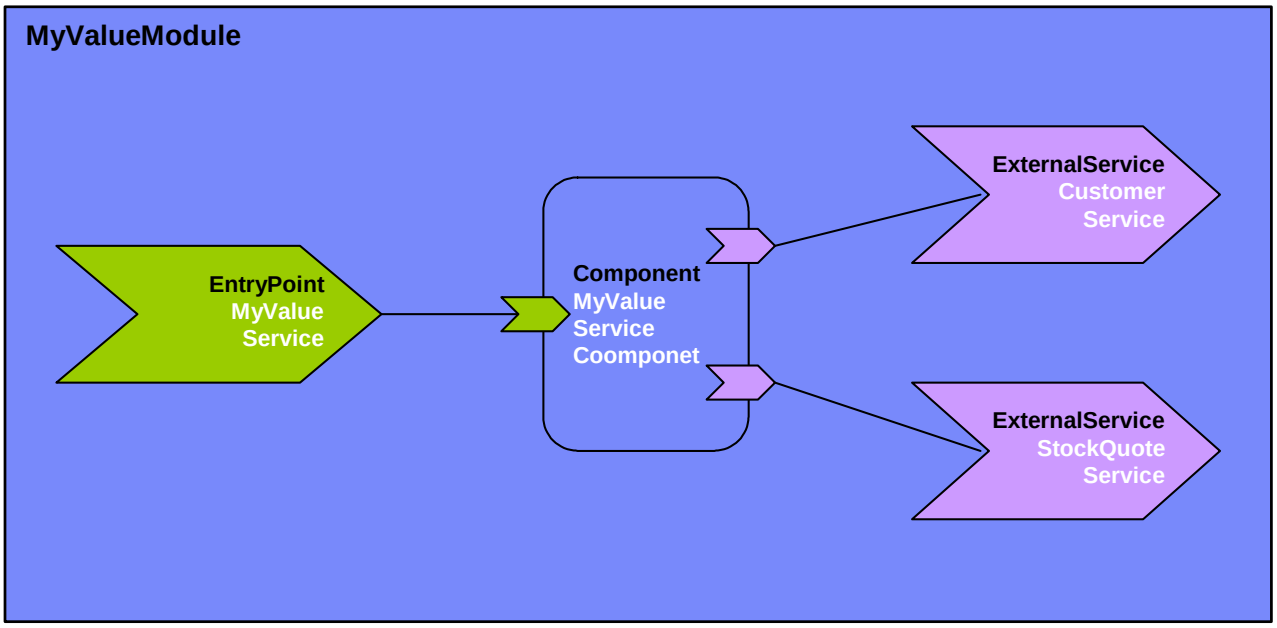


Figure 5: Module diagram for MyValueModule

The following snippet shows the sca.module file for the MyValueModule containing the component element for the MyValueServiceComponent. We see the setting of a property named currency, and the wiring of the customerService and stockQuoteService references.

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

   name="MyValueModule" >

   …

   <component name="MyValueServiceComponent">
         <implementation.java class="services.myvalue.MyValueServiceImpl"/>
         <properties>
               <v:currency>EURO</v:currency>
         </properties>
         <references>
            <v:customerService>CustomerService</v:customerService>
            <v:stockQuoteService>StockQuoteMediatorComponent</v:stockQuoteService>
         </references>
   </component>
   …

</module>
```

The following snippet gives an example of the layout of module file if both the currency property and the customerService reference of the MyValueServiceComponent had been declared to be multi-valued (many=true for the property and multiplicity=0..n or 1..n for the reference):

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

```
      xmlns:v="http://www.osoa.org/xmlns/sca/values/0.9"

   name="MyValueModule" >

   …

   <component name="MyValueServiceComponent">
      <implementation.java class="services.myvalue.MyValueServiceImpl"/>
      <properties>
         <v:currency>EURO</v:currency>
         <v:currency>Pound</v:currency>
         <v:currency>USDollar</v:currency>
      </properties>
      <references>
         <v:customerService>CustomerService1</v:customerService>
         <v:customerService>CustomerService2</v:customerService>
         <v:stockQuoteService>StockQuoteMediatorComponent</v:stockQuoteService>
      </references>
   </component>
   …

</module>
```

….this assumes that there are two external services named CustomerService1 and CustomerService2, both of which are wired to the customerService reference.

1.3.6 Override Attribute

It is possible for the assembler of a module to specify that some aspects of the module either may or must be overridden by the subsystem that deploys the module. The aspects that can be overridden are:

- component properties
- entry point bindings
- external service bindings

Entry point bindings are always overridable, since it must always be possible for subsystem to expose an entry point at a different address or binding than that listed in the module.

The module assembler has a number of options regarding whether external services or component properties may be overridden. Each of these elements has an ***override*** attribute with the following values:

- **no** – the value may not be overridden by the subsystem. This is the default for properties.
- **may** – the value may be overridden by the subsystem. This is the default for external services.
- **must** – the module does not specify a value and the subsystem must specify one.

Full details of the operation of Overrides are described in the Subsystem Overrides section.

1.3.7 External Services

SCA external services within a module represent remote services that are external to the SCA module that uses the service. These external services can be accessed by components within the module like any service provided by an SCA component.

External services use ***bindings*** to describe the access to external services. See the section on Bindings for more details of bindings.

External services are declared as child elements of a module contained in an ***sca.module*** file. An external service is represented by an ***externalService element*** which is a child of the module element. There can be ***zero or more*** external***Service*** elements in a module. The following snippet shows the module schema with the schema for a external***Service*** element.

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

   name="xs:NCName" >

   …

   <externalService name="xs:NCName" override="sca:OverrideOptions"?>*
         <interface.interface-type/>
         <binding.binding-type uri="xs:anyURI"/>*
   </externalService>

   …

</module>
```

The external***Service*** element has the following ***attributes***:

- ***name*** – the name of the external service. The name:
  - must be unique across all the external services in the module,
  - no external service can have the same name as a component in the same module since they both can be targets of wires
- ***override*** – whether the external service may be overridden. It has the following values:
  - **no** – the binding and endpoint address specified must remain unchanged. In this case, the external service does not appear as a reference on the moduleComponent at the subsystem level.
  - **may** – the service specified by the binding may be overridden by the subsystem. This is the default.
  - **must** – the subsystem must bind this external reference. There should be no ***binding*** element provided if this option is used.

An ***externalService*** has one service, which in turn has ***1 interface***. The interface is described by an ***interface element*** which is a child element of the ***externalService*** element. The interface can be specified using WSDL or a Java interface or another interface definition language - for further details on the interface element see the Interface section.

The interface of an external service must be remotable. The ***data exchange semantics*** between a client and an external service are always such that the implementation of the remote service may modify input messages (parameters) during or after an invocation and may modify return messages (results) after the invocation and no modification of input messages or post-invocation modifications to return messages are seen by the client.

An ***externalService*** element has zero or more ***binding elements*** as children. Details of the binding element are described in the Bindings section.

The ***endpoint address*** of an external service, which is specified as part of the child ***binding element***, may either be specified at design time or at deployment time. In addition, an endpoint address specified at design time may be overridden at deployment time, depending on the value of the ***override*** attribute, which is explained in the section on Overrides.

The external service becomes a reference of the module component. It may be wired at the subsystem level to a service of another module component or to an external service at the subsystem level, depending on the value of the override attribute.

An external service is a valid target of a reference ***wire***. The name of the external service is the value set on the reference targeting the external service.

1.3.7.1 Example External Service

The following figure shows the external service symbol that used to represent an external service in a module diagram.

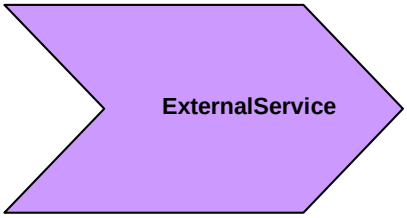


Figure 6: ExternalService symbol

The following figure shows the module diagram for the MyValueModule containing the external service CustomerService and the external service StockQuoteService.

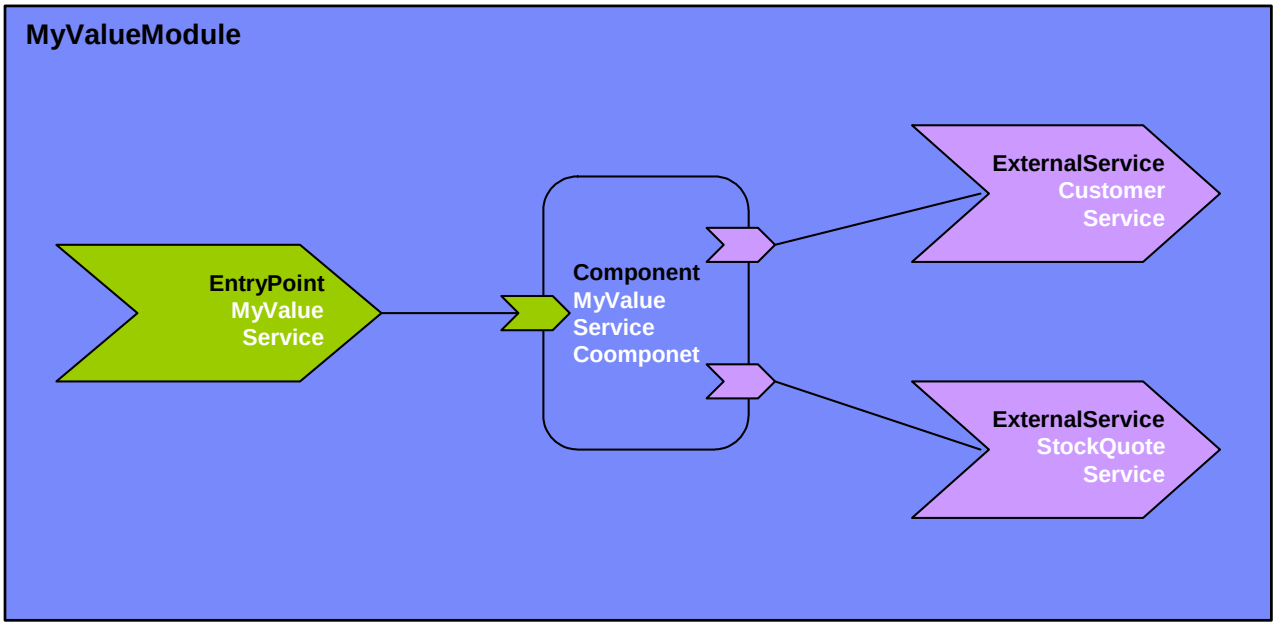


Figure 7: MyValueModule showing External Services

The following snippet shows the sca.module file for the MyValueModule file containing the external service elements for the CustomerService and the StockQuoteService. The external service CustomerService is bound using the SCA remote binding. The external service StockQuoteService is bound using the Web service binding. The endpoint addresses of the bindings can be specified, for example using the binding ***uri*** attribute (for details see the Bindings section), or overridden in an SCA subsystem configuration. Although in this case the external service StockQuoteService is bound to a Web service, its interface is defined by a Java interface, which was created from the WSDL portType of the web service.

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

   name="MyValueModule" >

   …

   <externalService name="CustomerService">
         <interface.java interface="services.customer.CustomerService"/>
         <binding.sca/>
   </externalService>

   <externalService name="StockQuoteService">
         <interface.java interface="services.stockquote.StockQuoteService"/>
         <binding.ws port="http://www.stockquote.org/StockQuoteService#
                     wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
   </externalService>

   …

</module>
```

1.3.8 Entry Point

Entry points are used to publish services provided by a ***module,*** so that they are addressable outside the boundaries of a module. The service published from the module can be a service of a component defined in the module, or an external service defined in the module. The latter case allows the republication an external service with a new address and/or new bindings.

An entry point publishes services provided by the SCA module using a specified ***binding***. See the section on Bindings for more details of bindings.

Entry points are defined as child elements of a module in an ***sca.module*** file. An entry point is represented by an ***entryPoint element*** which is a child of the module element. There can be ***zero or more*** entryPoint elements in a module. The following snippet shows the module schema with the schema for an entryPoint child element.

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

   name="xs:NCName" >


   <entryPoint name="xs:NCName" multiplicity="0..1 or 1..1 or 0..n or 1..n"?>*
         <interface.interface-type/>
```

```
        <binding.binding-type uri="xs:anyURI"/>+
        <reference>wire-target-URI</reference>
    </entryPoint>

    …

</module>
```

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

- ***name*** – the name of the entryPoint, the name is unique across all the entryPoints in the module, no entryPoint in the same module can have the same name as a component since they both can be sources of wires
- ***multiplicity (optional)*** - the multiplicity can have the following values
    - 1..1 (default) – one wire can have the entry point as a source
    - 0..1 – zero or one wire can have the reference as a source
    - 1..n – one or more wires can have the reference as a source
    - 0..n - zero or more wires can have the entry point as a source

Multiplicity defines the number of services that can be wired to the entryPoint:

- 1..1 is the simplest and most common case, where exactly one service is wired – the assembly is in error if no service is wired to the entryPoint.
- 0..1 caters for an implementation that can tolerate the case where no service is wired to the entryPoint, where correct operation of the system will occur even service is wired.
- 0..n and 1..n cater for the situation where the entryPoint can handle multiple target services. The two cases differ according to whether the implementation can tolerate no service being wired to the entryPoint – 0..n allows for no service to be wired while 1..n requires at least one. In both these cases, the ***collection*** of wired services get called than a single one.
  0..n is likely to find use in “publish/subscribe” event type relationships, where many services can receive a message, but where the implementation itself can operate successfully even if there are none.
  1..n might be used in situations where the implementation wants to ensure that at least one service gets the message.

An entry point has one reference which in turn has ***1 interface***. The interface is described by an ***interface element*** which is a child element of the entryPoint element. For details on the interface element see the interface section.

Since the Service represented by an Entry Point must be ***Remotable,*** arguments and returns of the interface used by an entry point must be translatable into the wire format specified by the binding. Implementations of remotable services may modify input messages (parameters) during or after an invocation and may modify return messages (results) after the invocation but no modification of input messages or post-invocation modifications to return messages are seen by the caller.

An entryPoint element has one or more ***binding elements*** as children. Details of the binding element are described in the Bindings section.

The entryPoint element has ***one reference element*** as its child. The value of the reference element ***wires*** the reference to a service that resolves the reference. For more details on reference wiring see the Wiring section.

An entry point definition in a module may be overridden at the subsystem level.

1.3.8.1 Entry Point Examples

The following figure shows the entry point symbol that used to represent an entry point in a module diagram.

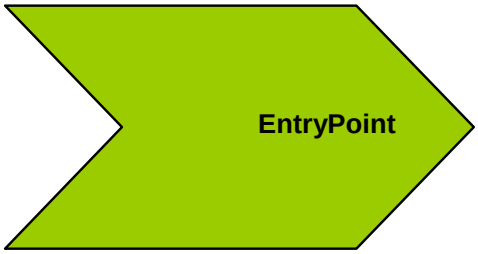


Figure 8: Entry Point symbol

The following figure shows the module diagram for the MyValueModule containing the entry point MyValueService.

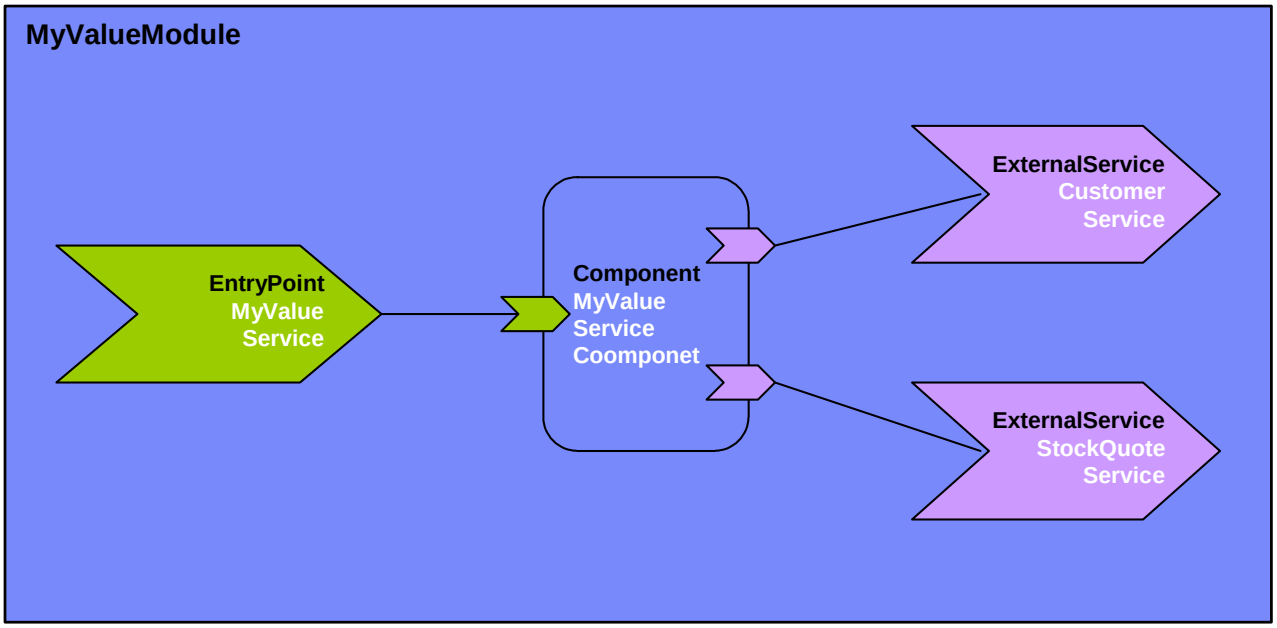


Figure 9: MyValueModule showing EntryPoint

The following snippet shows the sca.module file for the MyValueModule file containing the entryPoint element for the MyValueService. The entry point MyValueService is bound using a Web service binding, and wired to the MyValueServiceComponent that it publishes.

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

```
    name="MyValueModule" >

    <entryPoint name="MyValueService">
          <interface.java interface="services.myvalue.MyValueService"/>
          <binding.ws port="http://www.myvalue.org/MyValueService#
                      wsdl.endpoint(MyValueService/MyValueServiceSOAP)"/>
          <reference>MyValueServiceComponent</reference>
    </entryPoint>

    …

</module>
```

The following figure shows the module diagram for the CustomerModule containing the entry point CustomerService.

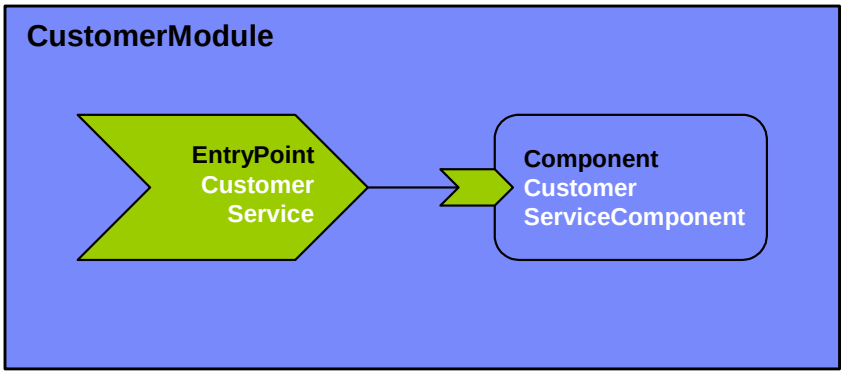


Figure 10: CustomerModule with CustomerService entry point

The following snippet shows the sca.module file for the CustomerModule file containing the entryPoint element for the CustomerService. The entry pointCustomerService is bound using the SCA remote binding, and wired to the CustomerServiceComponent that it publishes.

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

    name="CustomerModule" >


    <entryPoint name="CustomerService">
          <interface.java interface="services.customer.CustomerService"/>
          <binding.sca/>
          <reference>CustomerServiceComponent</reference>
    </entryPoint>

    …

</module>
```

1.3.9 Wire

SCA wires within a module connect ***service references*** to ***services***. Valid references are component references and entry points. Valid services are component services and external services.

If the component or entry point that is the source of the wire is defined in the same module file or the same module fragment file as the wire, then wires are defined by ***configuring references of components or entry points***. References are configured with the wire-target-URI of the service that resolves the reference.

If the component or entry point that is the source of the wire is defined in a different module file or in a different module fragment file then a Wire is represented by a ***wire element*** which is a child of the module element or moduleFragment element. There can be ***zero or more*** wire elements in a module or module fragment.

The following snippets shows the module schema with the schema for the reference elements of components and entry points, and the wire child element

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

   name="xs:NCName" >

   <entryPoint name="xs:NCName" multiplicity="0..1 or 1..1 or 0..n or 1..n"?>*
         <interface.interface-type/>
         <binding.binding-type uri="xs:anyURI"/>+
         <reference>wire-target-URI</reference>
   </entryPoint>

   <component name="xs:NCName">*
         <implementation.implementation-type/>
         <properties>?
               <v:property-name>property-value</v:property-name>+
         </properties>
         <references>?
               <v:reference-name>wire-target-URI</v:reference-name>+
         </references>
   </component>

   …

   <wire>*
         <source.uri>wire-source-URI</source.uri>
         <target.uri>wire-target-URI</target.uri>
   </wire>

</module>
```

The ***<reference-name> element of a component*** and the ***reference element of an entry point*** can have either of the following ***wire-target-URI*** values:

- <component-name>/<service-name>
    - the specification of the service name is optional if the target component only has one service with a compatible interface
- <externalReference-name>

The wire has the following child elements:

- ***source*** – names the component or entry point, valid URI schemes are
  - <component-name>/<reference-name>
    - the specification of the reference name is optional if the source component only has one reference
  - <entryPoint-name>
- ***target*** – name the component or external service, valid URI schemes are
  - <component-name>/<service-name>
    - the specification of the service name is optional if the target component only has one service with a compatible interface
  - <externalReference-name>

Wires can only link references and services that are contained in the same module (irrespective of which file or files are used to describe the module). Wiring to entities outside the module is done through entry points and external services.

A wire may only connect a reference to a service if the service implements an interface that is compatible with the interface required by the reference**.** The reference and the service are compatible if:

1. the service interface and the reference interface are either both remotable or they are both local
2. the methods on the target service interface are a superset of the methods in the interface specified on the reference.
3. compatibility for the individual method is defined as compatibility of the signature, that is method name, input types, and output types have to be the same.
4. the order of the input and output types must also be the same.
5. the set of Faults and Exceptions expected by the reference should be the same or be a superset of those specified by the service.
6. other specified attributes of the two interfaces must match, including Scope and Callback interface

A Wire can connect between different interface languages (eg. Java interfaces and WSDL portTypes) in either direction, as long as the operations defined by the two interface types are equivalent. They are equivalent if the operation(s), parameter(s), return value(s) and faults/exceptions map to each other.

Service clients cannot (portably) ask questions at runtime about additional interfaces that are provided by the implementation of the service (e.g. the result of “instance of” in Java is non portable). Some vendors may choose to have proxies for all wires.

Note: It is permitted to deploy a module that has references that are not wired. For the case of an un-wired reference with multiplicity 1..1 or 1..n the deployment process provided by an SCA run-time should issue a warning.

1.3.9.1 Wire Examples

The following figure shows the module diagram for the MyValueModule containing wires between entry points, components, and external services.

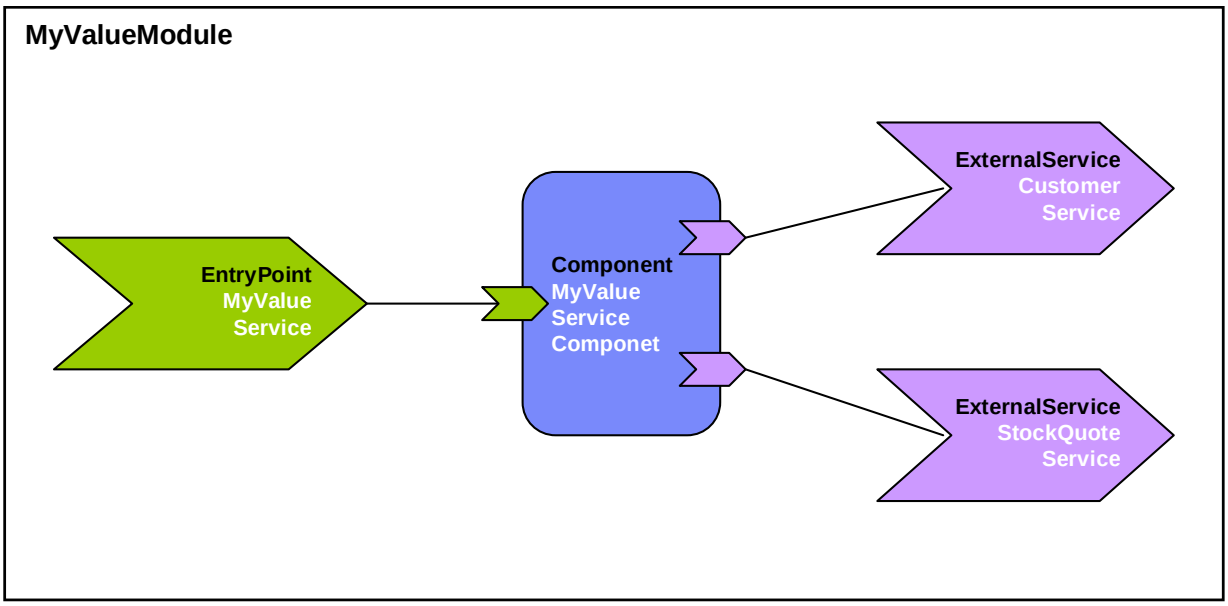


Figure 11: MyValueModule showing Wires

The following snippet shows the sca.module file for the MyValueModule containing the configured component and entry point references. The entry point MyValueService is wired to the MyValueServiceComponent. The MyValueServiceComponent's customerService reference is wired to the external service CustomerService. The MyValueServiceComponent's stockQuoteService reference is wired to the external service StockQuoteService.

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

   name="MyValueModule" >

   <entryPoint name="MyValueService">
         <interface.java interface="services.myvalue.MyValueService"/>
         <binding.ws port="http://www.myvalue.org/MyValueService#
                     wsdl.endpoint(MyValueService/MyValueServiceSOAP)"/>
         <reference>MyValueServiceComponent</reference>
   </entryPoint>

   <component name="MyValueServiceComponent">
         <implementation.java class="services.myvalue.MyValueServiceImpl"/>
         <properties>
               <v:currency>EURO</v:currency>
         </properties>
         <references>
               <v:customerService>CustomerService</v:customerService>
               <v:stockQuoteService>StockQuoteService</v:stockQuoteService>
         </references>
   </component>

   <externalService name="CustomerService">
         <interface.java interface="services.customer.CustomerService"/>
         <binding.sca/>
```

```
    </externalService>

    <externalService name="StockQuoteService">
          <interface.java interface="services.stockquote.StockQuoteService"/>
          <binding.ws port="http://www.stockquote.org/StockQuoteService#
                        wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
    </externalService>

</module>
```

1.3.10 Module Fragments

In order to assist team development, modules may be decomposed into multiple physical artifacts that are merged into a single logical unit at deployment. While a module is defined in an ***sca.module*** file, the module may receive additional configuration through ***module fragment*** files. Module fragment files are identified with a ***.fragment*** suffix and must be located alongside their associated ***sca.module*** file (that is, in the same file system or archive directory).

The semantics of module fragments are that they are inlined into the base **sca.module** file; they may therefore contain components, entry points, external service, and wires. Moreover, named artifacts defined in an ***sca.module*** or in another associated module fragment may be referenced. For example, it is permissible to have two components in an ***sca.module*** with a wire specifying one as the source and the other as the target in an associated module fragment.

It is an error if there are duplicated elements in the combined module (eg. two entry points with the same uri contributed by different module fragments).

Module Fragments are defined by a ***moduleFragment element*** in a ***.fragment*** file. The following snippet shows the moduleFragement schema.

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

    name="xs:NCName" >

    <entryPoint name="xs:NCName" multiplicity="0..1 or 1..1 or 0..n or 1..n"?>*
          <interface.interface-type/>
          <binding.binding-type uri="xs:anyURI"/>+
          <reference>wire-target-URI</reference>
    </entryPoint>

    <component name="xs:NCName">*
          <implementation.implementation-type/>
          <properties>?
                <v:property-name>property-value</v:property-name>+
          </properties>
          <references>?
                <v:reference-name>wire-target-URI</v:reference-name>+
          </references>
    </component>

    <externalService name="xs:NCName">*
          <interface.interface-type/>
          <binding.binding-type uri="xs:anyURI"/>*
    </externalService>

    <wire>*
```

```
        <source.uri>wire-source-URI</source.uri>
        <target.uri>wire-target-URI</target.uri>
    </wire>

</moduleFragment>
```

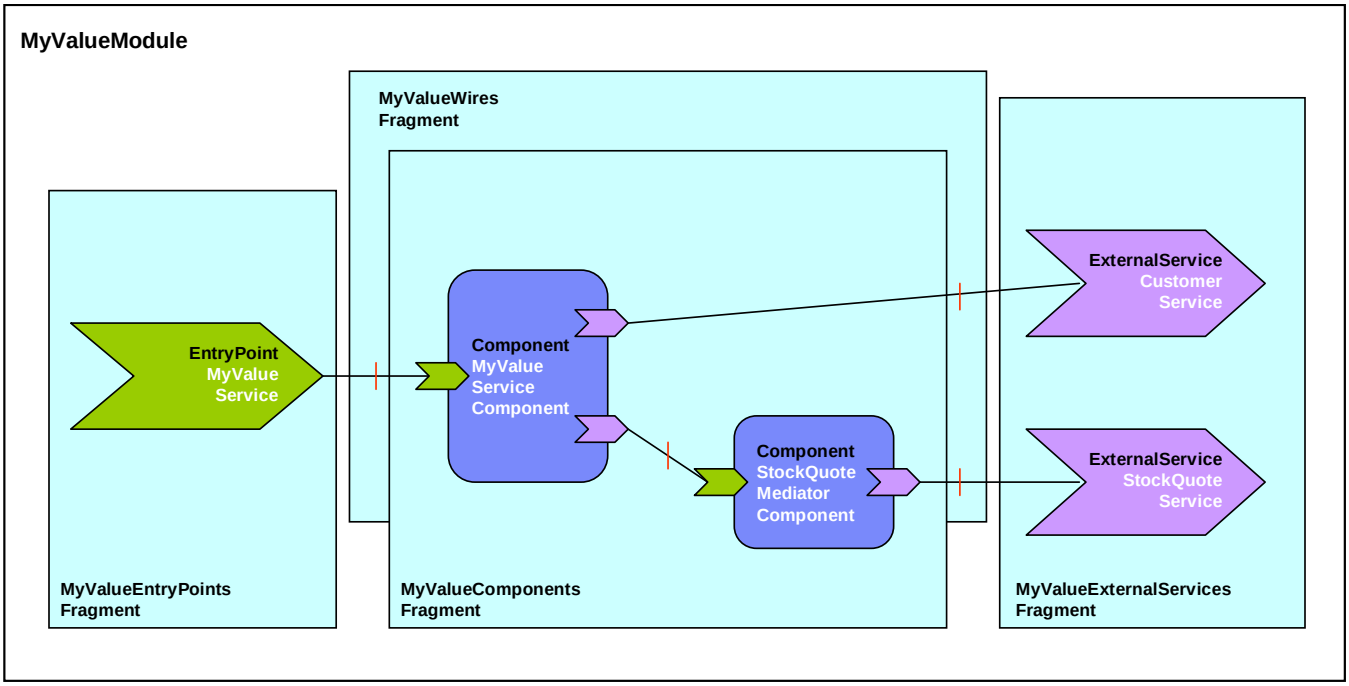
The moduleFragment element has the following ***attribute***:

- ***name*** – the name of the moduleFragment. Note that this name is not used when wiring elements within the module.

The moduleFragment element has the same child elements as the module element in the sca.module file. For the definition of the elements see the previous sections.

1.3.10.1 ModuleFragment Examples

The following figure shows the module diagram for the MyValueModule containing four module fragments. The ***MyValueEntryPoints fragment*** contains the MyValueService entry point. The ***MyValueComponents fragment*** contains the MyValueServiceComponent and the StockQuoteMediatorComponent as well as the wire between them. The ***MyValueExternalReferences fragment*** contains the CustomerService and StockQuoteService external service. The ***MyValueWires fragment*** contains the wires that connect the MyValueService entry point to the MyValueServiceComponent, that connect the customerService reference of the MyValueServiceComponent to the CustomerService external service, and that connect the stockQuoteService reference of the StockQuoteMediatorComponent to the StockQuoteService external reference. Note that this is just one way of fragmenting the MyValueModule.



The following snippet shows the contents of the sca.module for the fragmented MyValueModule. In this sample it only provides the name of the module. As said earlier also the module file itself can be used in a fragmented module scenario to define components, external services, entry points and wires.

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

   name="MyValueModule" >

</module>
```

The following snippet shows the content of the MyValueEntryPoints.fragment file.

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

   name="MyValueEntryPoints" >

   <entryPoint name="MyValueService">
         <interface.java interface="services.myvalue.MyValueService"/>
         <binding.ws port="http://www.myvalue.org/MyValueService#
                     wsdl.endpoint(MyValueService/MyValueServiceSOAP)"/>
   </entryPoint>

</moduleFragment>
```

The following snippet shows the content of the MyValueServiceComponents.fragment file.

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

   name="MyValueServiceComponents" >

   <component name="MyValueServiceComponent">
         <implementation.java class="services.myvalue.MyValueServiceImpl"/>
         <properties>
               <v:currency>EURO</v:currency>
         </properties>
         <references>
               <v:stockQuoteService>
                     StockQuoteMediatorComponent
               </v:stockQuoteService>
         </references>
   </component>

   <component name="StockQuoteMediatorComponent">
         <implementation.java class="services.mediator.StockQuoteMediatorImpl"/>
   </component>
```

```
</moduleFragment>
```

The following snippet shows the content of the MyValueExternalReferences.fragment file.

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

   name="MyValueExternalReferences" >

   <externalService name="CustomerService">
         <interface.java interface="services.customer.CustomerService"/>
         <binding.sca/>
   </externalService>

   <externalService name="StockQuoteService">
         <interface.java interface="services.stockquote.StockQuoteService"/>
         <binding.ws port="http://www.stockquote.org/StockQuoteService#
                     wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
   </externalService>

</moduleFragment>
```

The following snippet shows the content of the MyValueWires.fragment file.

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

   name="MyValueWires" >

   <wire>
         <source.uri>MyValueService</source.uri>
         <target.uri>MyValueServiceComponent</target.uri>
   </wire>
   <wire>
         <source.uri>MyValueServiceComponent/customerService</source.uri>
         <target.uri>CustomerService</target.uri>
   </wire>
   <wire>
         <source.uri>StockQuoteMediatorComponent</source.uri>
         <target.uri>StockQuoteService</target.uri>
   </wire>

</moduleFragment>
```

1.4 Subsystem

An ***SCA Subsystem*** is used to group modules which provide related business function, through the configuration and administration of module components, external services and entry points, and the wires that interconnect them in a ***SCA system*** (i.e. the complete SCA runtime). The configuration of an SCA system is represented by the combination of all the Subsystems deployed into it.

The module components in a subsystem configuration are components that are implemented by SCA modules and which have services and references.

Subsystems are defined by a subsystem element in a ***sca.subsystem*** file. The following snippet shows the system schema.

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

   name="xs:NCName" uri="xs:anyURI">

   <entryPoint name="xs:NCName" multiplicity="0..1 or 1..1 or 0..n or 1..n"?>*
         <interface.interface-type/>
         <binding.binding-type uri="xs:anyURI"/>+
         <reference>wire-target-URI</reference>?
   </entryPoint>

   <moduleComponent name="xs:NCName" uri="xs:anyURI" module="xs:NCName">*
         <properties>?
               <v:property-name>property-value</v:property-name>+
         </properties>
         <references>?
               <v:reference-name>wire-target-URI</v:reference-name>
         </references>
   </moduleComponent>

   <externalService name="xs:NCName">*
         <interface.interface-type/>
         <binding.binding-type uri="xs:anyURI"/>+
   </externalService>

   <wire>*
         <source.address-type/>
         <target.address-type/>
   </wire>

</subsystem>
```

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

- ***name*** – the name of the subsystem, which must be unique in an SCA system
- ***uri*** – The URI to use as a prefix of entry points and of module component services contained in this subsystem. If it is relative, it is relative to the base URI of the system. It defaults to a relative URI of the subsystem ***name*** attribute.

Note: The Eclipse naming convention for plugins provides a good way to achieve unique names, e.g. com.mycompany.myvaluesubsystem . This format is recommended but is not normative.

The child elements of the subsystem element are described in detail in the following sections – moduleComponent, entryPoint, externalReference and wire.

1.4.1 Module Component

A ***module component*** is a configured module within a subsystem. The module referenced by a module component is said to ***implement*** the module component. The services provided by a module component are defined by the entry points in the module which implements the module component. The references of a module component are defined by the external services of the module which implements the module component.

In contrast to services and references of components within a module, which are only typed by interface, services and references of module components are typed by interface and binding as specified by the entry points and external services that define them.

Module components are part of a subsystem contained in an ***sca.subsystem*** file. A module component is represented by a ***moduleComponent element*** which is a child of the subsystem element. There can be ***zero or more*** moduleComponent elements in a subsystem. The following snippet shows the subsystem schema with the schema for the moduleComponent child element.

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

   name="xs:NCName" uri="xs:anyURI" >

   …

   <moduleComponent name="xs:NCName" uri="xs:anyURI" module="xs:NCName">*
         <properties>?
               <v:property-name>property-value</v:property-name>+
         </properties>
         <references>?
               <v:reference-name>wire-target-URI</v:reference-name>
         </references>
   </moduleComponent>

   …

</subsystem>
```

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

- ***name*** – the name of the module component, the name has to be unique across all module components of the SCA subsystem, no entry point in the subsystem can have the same name as a module component since they both can be sources of wires, no external service in the subsystem can have the same name as a module component since they both can be targets of wires
- ***uri*** – URI to use as the prefix of entry points contained in this module. If it is relative, it is relative to the URI of the subsystem that the module component is contained in. It defaults to a relative URI of the moduleComponent ***name*** attribute.

- ***module*** – the name of the module implementing the subsystem

The module component element can optionally have a ***properties*** child element. The properties that may be specified are the overrideable properties from the module which implements the moduleComponent. A property in the module whose override attribute is **must** must be specified in the module component. See the Override section for more details.

The module component element can have ***zero or one references element*** as its child. The references element has one or more **<reference-name> elements** as its children. The references that can be configured are defined by the module. The value of the <reference-name> element ***wires*** the reference to a service that resolves the reference. For more details on reference wiring see the wiring section.

1.4.2 Overrides

It is possible for the assembler of a module to specify that some aspects of the module either may or must be overridden by the subsystem that deploys the module. The aspects that can be overridden are:

- component properties
- entry point bindings
- external service bindings.

Entry point bindings are always overridable, since it should always be possible for subsystem to expose an entry point at a different address or binding than that listed in the module.

The module assembler has a number of options regarding whether external services or component properties may be overridden. Each of these elements has an ***override*** attribute with the following values:

- **no** – the value may not be overridden by the subsystem. This is the default for properties.
- **may** – the value may be overridden by the subsystem. This is the default for external services.
- **must** – the module does not specify a value and the subsystem must specify one.

An external service of a module is overridden in a subsystem by wiring the moduleComponent's corresponding reference to a different target service – this can be restricted to simply defininig a new endpoint address for the external service. An entry point of a module is overridden by subsystem-level wiring which specifies the required bindings.

A component property that is overridable becomes a property of the module that contains it. The property is overridden by specifying a value for the property on the ***moduleComponent*** of the subsystem that deploys it. The type of the moduleComponent property is determined by the type of the overridable property. The name of the property defaults to the name of the property for the component. However, since multiple components may contain properties with the same name, it may be necessary to specify a different name for the moduleComponent property. The component property element may include a ***modulePropertyName*** attribute whose value is used as the name of the property of the moduleComponent.

1.4.2.1 ModuleComponent Examples

The following figure shows the ***module component symbol*** that we use to represent a subsystem in a system diagram.

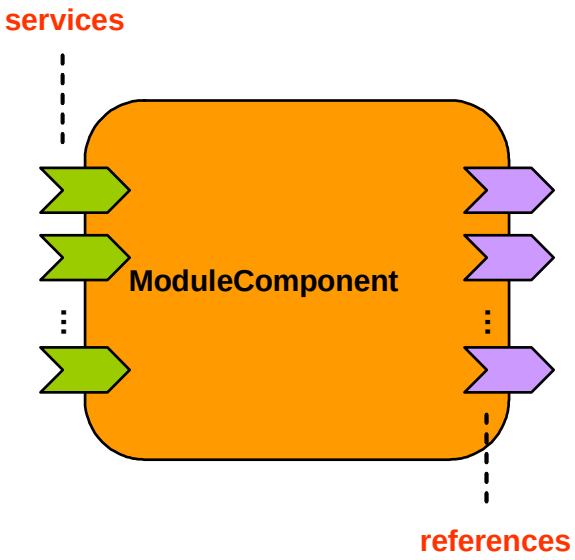


Figure 12: ModuleComponent symbol

The following figure shows the subsystem diagram for the MyValueSubsystem containing the MyValueModuleComponent and the CustomerModuleComponent. The module components show the services and references according to the entry points and external services of the modules that implement the module components.

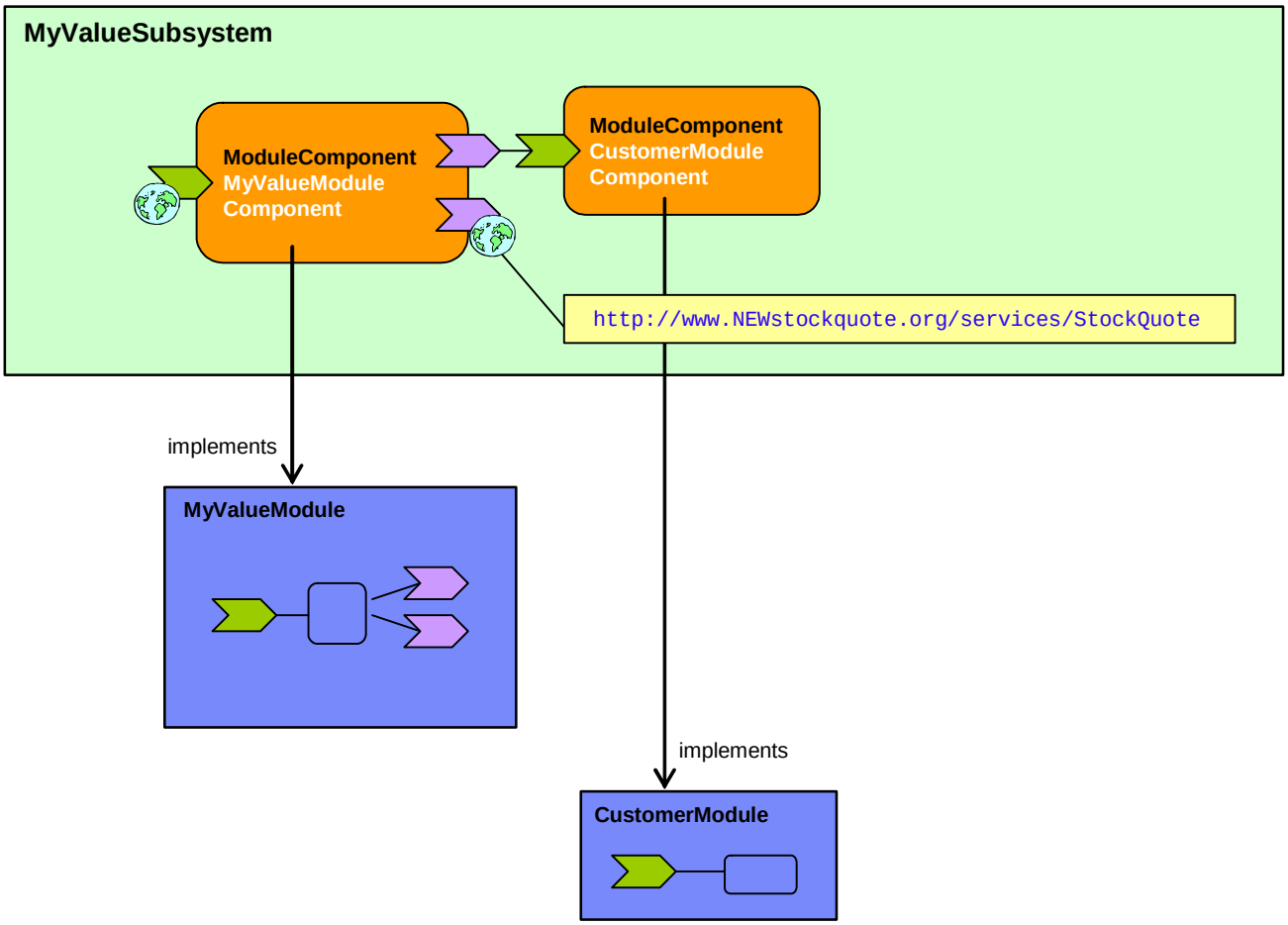


Figure 13: Subsystem diagram for MyValueSubsystem

The following snippet shows the sca.subsystem file for the MyValueSubsystem containing the moduleComponent element for the MyValueModuleComponent and the CustomerModuleComponent. The wiring of the CustomerService and StockQuoteService references is shown, with the references of the MyValueModuleComponent overriding the endpoint addresses defined within MyValueModule:

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

   name="MyValueSystem" >


   <moduleComponent name="MyValueModuleComponent" module="MyValueModule">
      <references>
         <v:CustomerService>CustomerModuleComponent/CustomerService
         </v:CustomerService>
         <v:StockQuoteService>http://www.NEWstockquote.org/services/StockQuote
         </v:StockQuoteService>
      </references>
   </moduleComponent>

   <moduleComponent name="CustomerModuleComponent" module="CustomerModule"/>
```

```
</subsystem>
```

1.4.3 External Service

External services represent remote services that are external to the SCA system that uses the service.

The differences between an external service on the module level and on the system level are:

- the name has to be unique across all external services defined in the subsystem. No external service in a subsystem can have the same name as a module component since they both can be targets of wires

External services are part of a subsystem contained in an ***sca.subsystem*** file. An external service is represented by an ***externalService element*** which is a child of the subsystem element. There can be ***zero or more*** externalService elements in a system. The following snippet shows the subsystem schema with the schema for an externalService child element.

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

   name="xs:NCName" uri="xs:anyURI" >

   …

   <externalService name="xs:NCName">*
         <interface.interface-type/>
         <binding.binding-type uri="xs:anyURI"/>+
   </externalService>

   …

</subsystem>
```

A subsystem that just contains one or more external services can be deployed into an SCA system.

An external service at the subsystem level can be used to override the bindings and/or the target address of a module component reference, where the ***override*** attribute of the implementing module's external service is set to ***may*** or ***must***. The binding(s) and target address(es) defined by the external service get applied when the external service is wired to the module component reference.

1.4.3.1 External Service Examples

The following figure shows the external service symbol that we use to represent an external service in a subsystem diagram.

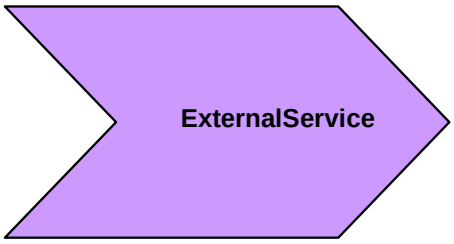


Figure 14: ExternalService symbol

The following figure shows the subsystem diagram for the StockQuoteServiceERSubsystem containing the external service StockQuoteServiceER.

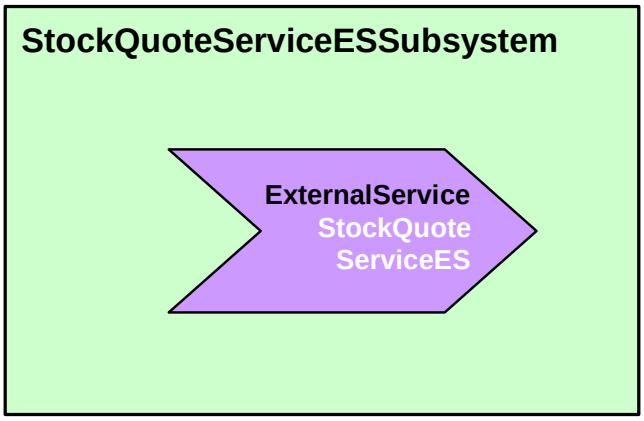


Figure 15: StockQuoteServiceESSubsystem diagram showing external service

The following snippet shows the sca.subsystem file for the StockQuoteServiceERSubsystem file containing the externalReference element for the StockQuoteServiceER.

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

   name="StockQuoteServiceSystem" >

   <externalService name="StockQuoteServiceER">
         <interface.wsdl   interface="http://www.stockquote.org/StockQuoteService#
                           wsdl.interface(StockQuote)"/>
         <binding.ws port="http://www.stockquote.org/StockQuoteService#
                     wsdl.endpoint(StockQuote/StockQuoteServiceSOAP)"/>

   </externalService>

</subsystem>
```

1.4.4 Entry Point

Entry points are used to declare the externally accessible services of a ***subsystem***.

The differences between entry points on the module level and on the system level are:

- the name has to be unique across all entry points defined in the subsystem. No entry point in the subsystem can have the same name as a module component since they both can be sources of wires
- the specification of the reference child element is optional, since it can be wired through a wire supplied by another subsystem.

Entry points are part of a subsystem contained in an ***sca.subsystem*** file. An entry point is represented by an ***entryPoint element*** which is a child of the subsystem element. There can be ***zero or more*** entryPoint elements in a subsystem. The following snippet shows the subsystem schema with the schema for a entryPoint child element.

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

   name="xs:NCName" uri="xs:AnyURI" >

   <entryPoint name="xs:NCName" multiplicity="0..1 or 1..1 or 0..n or 1..n"?>*
         <interface.interface-type/>
         <binding.binding-type uri="xs:AnyURI"/>+
         <reference>wire-target-URI</reference>?
   </entryPoint>

   …

</subsystem>
```

You can deploy a subsystem into an SCA system that just contains one or more entry points.

A module entry point is overridden by wiring the corresponding moduleComponent service to a subsystem-level entry point that has the same binding URI as the module component's entry point after the URI expansion rules are applied. In this case, the system-level entry point completely replaces the corresponding binding specified in the module-level entry point. If a system entry point has a binding whose URI differs from the entry point of the target module component, then the system entry point provides a binding that is in addition to the binding provided by the module component.

1.4.4.1 Entry Point Examples

The following figure shows the entry point symbol that we use to represent an entry point in a subsystem diagram.

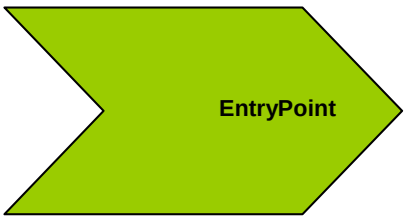


Figure 16: EntryPoint symbol

The following figure shows the system diagram for the StockQuoteServiceEPSubsystem containing the entry point StockQuoteServiceEP.

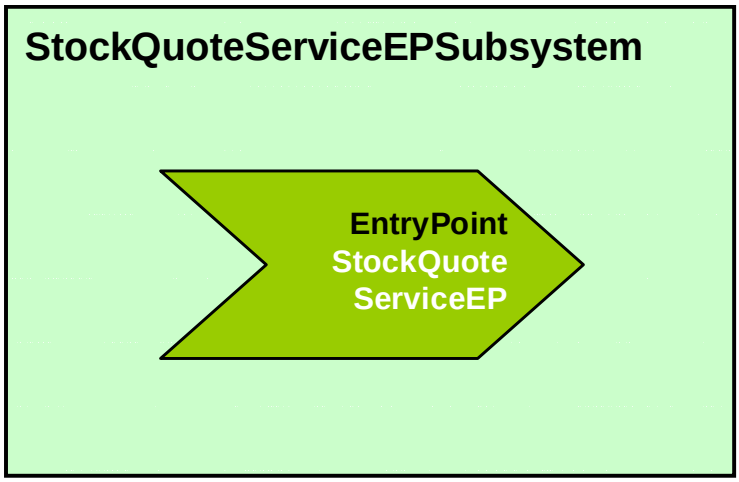


Figure 17: StockQuoteServiceEPSubsystem diagram showing entry point

The following snippet shows the sca.subsystem file for the StockQuoteServiceEPSubsystem file containing the externalProvider element for the StockQuoteServiceEP.

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

   name="StockQuoteServiceEPSubsystem" >

   <entryPoint name="StockQuoteServiceEP">
         <interface.wsdl
         interface="http://www.sq.org/SQService#wsdl.interface(SQService)"/>
         <binding.ws
   port="http://www.sq.org/SQService#wsdl.endpoint(SQService/SQServiceSOAP)"/>
   </entryPoint>

</subsystem>
```

1.4.5 Wire

SCA Wires within a subsystem connect ***external services*** to ***entry points***.

The differences between wires on the module level and on the system level are:

- The sources and targets of the wires don't have to be part of the subsystem that contains the wire. The sources and targets can be defined by other subsystems or can be external to the SCA system.
- A sources and a target can only be wired if their binding is compatible. This is not an issue inside a module since all bindings are local.
- It is possible to have subsystems that just contain wires.

Wires are defined in a subsystem contained in an sca.susbsytem file in ***one of two ways***.

If the module component or entry point that is the source of the wire is defined in the same subsystem as the wire, then wires are defined by ***configuring references of module components or entry points***. References are configured with the wire-target-URI of the service that resolves the reference.

If the module component or entry point that is the source of the wire is defined in a different subsystem than the wire, then a wire is represented by a ***wire element*** which is a child of the subsystem element. There can be ***zero or more*** wire elements in a subsystem. In addition, where the source or target of the wire is in a different subsystem from the wire, the name of the source/target must be qualified with the name of the subsystem.

The following snippets shows the subsystem schema with the schema for the reference elements of module components and entry points, and the wire child element

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

   name="xs:NCName" uri="xs:anyURI" >

   <entryPoint name="xs:NCName" multiplicity="0..1 or 1..1 or 0..n or 1..n"?>*
         <interface.interface-type/>
         <binding.binding-type uri="xs:anyURI"/>+
         <reference>wire-target-URI</reference>?
   </entryPoint>

   <moduleComponent name="xs:NCName" uri="xs:anyURI" module="xs:NCName">*
         <references>?
               <v:reference-name>wire-target-URI</v:reference-name>
         </references>
   </moduleComponent>

   …

   <wire>*
         <source.address-type/>
         <target.address-type/>
   </wire>

</subsystem>
```

The ***<reference-name> element of a module component*** or the ***reference element of an entry point*** can have the following ***wire-target-URI*** values:

- <moduleComponent-name>/<service-name>
    - the specification of the service name is optional if the source component only has one
- <externalReference-name>
- or a binding specific endpoint address
    - e.g. http://www.NEWstockquote.org/services/StockQuote

The wire has the following child elements:

- ***source*** – names the module component or entry point, valid URI schemes are
  - <moduleComponent-name>/<reference-name>
    - the specification of the reference name is optional if the source module component only has one reference
  - <entryPoint-name>
  - or a binding specific endpoint address
    - e.g. topic://Quotes
- ***target*** – names the module component or external service, valid URI schemes are
  - <moduleComponent-name>/<service-name>
    - the specification of the service name is optional if the source module component only has one service which has a compatible interface
  - <externalReference-name>
  - or a binding specific endpoint address
    - e.g. http://www.stockquote.org/services/StockQuote

1.4.5.1 Wire Examples

The following figure shows the subsystem diagram for the MyValueSubsystem containing the MyValueModuleComponent, the CustomerModuleComponent, and the wire that connects them.

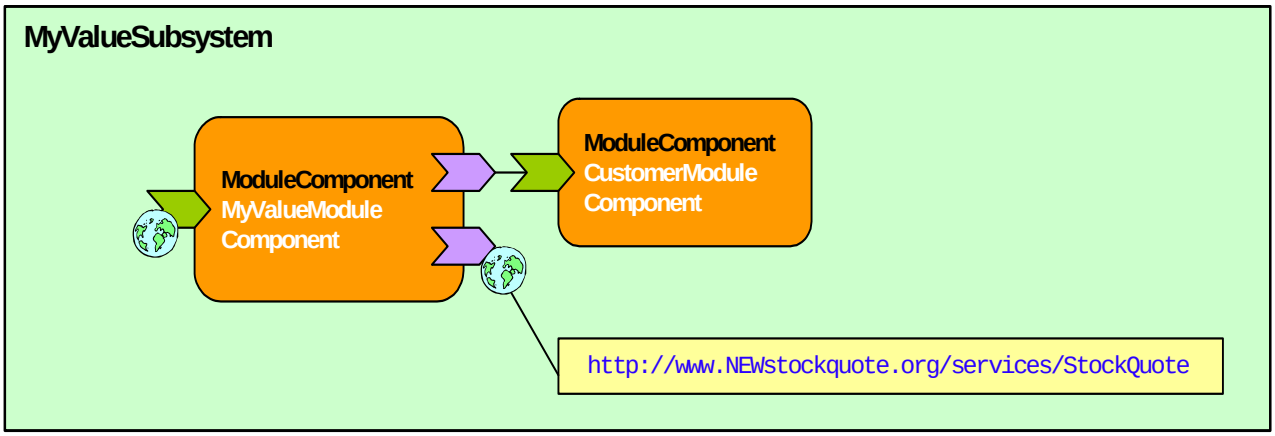


Figure 18: MyValueSubsystem diagram showing wires

The following snippet shows the sca.subsystem file for the MyValueSubsystem containing the moduleComponent element for the MyValueModuleComponent and the CustomerModuleComponent. The CustomerService reference of the MyValueModuleComponent is wired to the CustomerService of the CustomerModuleComponent. The StockQuoteService reference which has a Web service binding is wired to a service that is external to the SCA system. Note that the endpoint address specified in the external service that defines this reference is overridden by the value specified in the moduleComponent reference element.

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

   name="MyValueSystem" >

   <moduleComponent name="MyValueModuleComponent" module="MyValueModule">
      <references>
         <v:CustomerService>
               CustomerModuleComponent/CustomerService
         </v:CustomerService>
         <v:StockQuoteService>
               http://www.NEWstockquote.org/services/StockQuote
         </v:StockQuoteService>
      </references>
   </moduleComponent>

   <moduleComponent name="CustomerModuleComponent" module="CustomerModule"/>

</subsystem>
```

In the next sample we wire the entry point StockQuoteServiceEP of the StockQuoteServiceEPSubsystem to the external service StockQuoteServiceER of the StockQuoteServiceERSubsystem. They can't be wired directly since their interfaces don't match. Two additional subsystems are created. The first is the StockQuoteMediatorSubsystem which provides the StockQuoteMediator module component that mediates between the mismatched interfaces. The second is the StockQuoteWireSubsystem that provides the wires to connect the other three together.

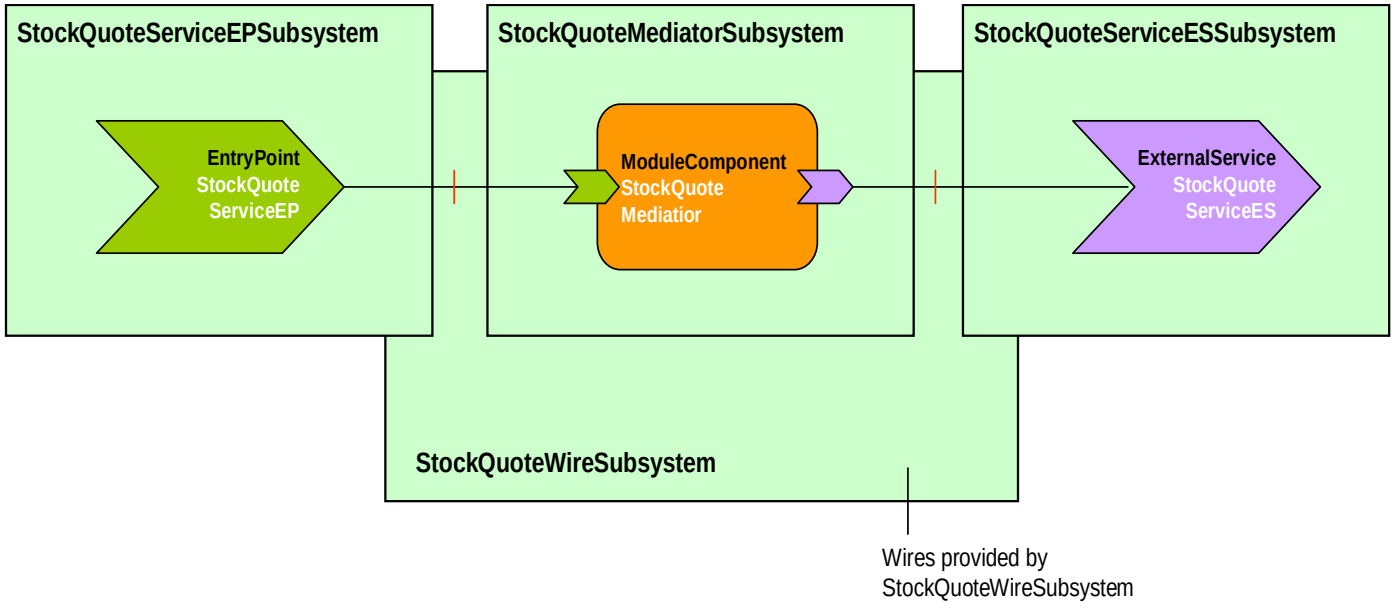


Figure 19: System diagram composed from multiple Subsystems

The following snippet shows the sca.subsystem file for the StockQuoteMediatorSubsystem containing the moduleComponent element for the StockQuoteMediator.

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

   name="StockQuoteMediatorSubsystem" >

   <moduleComponent name="StockQuoteMediator" module="StockQuoteMediatorModule"/>

</subsystem>
```

The following snippet shows the sca.subsystem file for the StockQuoteWireSubsystem containing two wires. The first connects the entry point StockQuoteServiceEP of the StockQuoteServiceEPSubsystem to the StockQuoteMediator of the StockQuoteMediatorSubsystem. The second connects the StockQuoteMediator of the StockQuoteMediatorSubsystem to the external service StockQuoteServiceER of the StockQuoteServiceER.

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

   name="StockQuoteWireSubsystem" >

   <wire>
         <source.uri>StockQuoteServiceEP</source.uri>
         <target.uri>StockQuoteMediator</target.uri>
   </wire>
   <wire>
         <source.uri>StockQuoteMediator</source.uri>
         <target.uri>StockQuoteServiceER</target.uri>
   </wire>

</subsystem>
```

1.5 Binding

Bindings are used by external services and entry points. External services use bindings to describe the access mechanism used to call an external service (which can be a service provided by another SCA module). Entry points use bindings to describe the access mechanism that clients (which can be a client from another SCA module) have to use to call the service published by the entry point.

SCA supports the use of multiple different types of bindings. Examples include ***SCA service, Web service, stateless session EJB, data base stored procedure, EIS service***. An SCA runtime must provide support for SCA service and Web service binding types. SCA provides an extensibility mechanism by which an SCA runtime can add support for additional binding types. For details on how additional binding types are defined, see the section on the Extension Model.

A binding is defined by a ***binding element*** which is a child element of an external service or entry point element in module file or in a subsystem file. The following snippet shows the module schema with the schema for the binding element.

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

   name="xs:NCName" >

   <entryPoint name="xs:NCName" >*
         <interface.interface-type/>
         <binding.binding-type uri="xs:anyURI"/>+
         <reference>wire-target-URI</reference>
   </entryPoint>

   …

   <externalService name="xs:NCName" override="sca:OverrideOptions”>*
         <interface.interface-type/>
         <binding.binding-type uri="xs:anyURI"/>*
   </externalService>

</module>
```

The name of the binding element is architected; it is in itself a qualified name. The first qualifier is always named “binding”, and the second qualifier names the respective binding-type (e.g. binding.sca, binding.ws, binding.ejb, binding.eis).

A binding element has an optional uri ***attribute*** with the following semantic.

- For an ***external service*** the URI attribute defines the wire-target URI of the external service. It is optional for external services defined in modules, but required for external services defined in subsystems. The URI attribute of an external service of a module can be reconfigured by the module component using the module. Some binding types may require that the address of the target service uses more than a simple URI (such as a WS-Addressing endpoint reference). In those cases, the binding type will define the additional attributes or sub-elements that are necessary to identify the service.

- For an ***entry point*** the URI attribute defines the URI relative to the module component. The default value for the URI is the name of the entry point.

When multiple bindings exist for an entry point, it means that the service is available by any of the specified bindings. The technique that the system uses to choose among available bindings is left to the implementation and it may include additional (nonstandard) configuration. Whatever technique is used should be documented.

Entry points can always have their bindings overridden at the subsystem level. External references are overridable only if the external reference has an ***override*** attribute of **may** or **must**. See the Override section for more details.

The following sections describe the SCA and Web service binding type in detail.

1.5.1 Form of the URI of a Deployed Binding

The effective URI that is used for a binding is made up of combination ***uri*** attributes as follows:

Base System URI for a scheme / Subsystem URI / Module Component URI / Binding URI

For an entry point where the Base URI is "http://acme.com", the subsystem name is "equitiesSubsystem", the module component is named "stocksComponent" and the entry point binding has a relative URI of "getQuote", the URI would look like this:

http://acme.com/equitiesSubsystem/stocksComponent/getQuote

The URI specified at each level may be absolute or relative. The path is shortened to only include the lowest (rightmost) absolute URI specified and all the relative URI that follow it. So, if the module component in the previous example had an absolute URI of "http://acme.com" rather than the relative URI of “stocksComponent” it has above, then the effective URI would end up being just:

http://acme.com/getQuote

The system may specify a base URI for any URI scheme that supports hierarchical URI. The URI constructed is only created for URI schemes for which the system has a base URI.

For any SCA element that has both a name and a uri attribute, the uri defaults to being the same as the name. If the ***binding*** element does not specify a uri attribute, the URI defaults to the name of the entry point that it is part of. Therefore, if no uri attributes are specified, the default URI for binding would be:

Base System URI / Subsystem name / Module Component name / Entry point name

Allowing a relative URI to be specified that differs from the name of a subsystem, module component or entry point name allows the URI hierarchy of entry points to be designed independently of the organization of the system.

It is good practice to design the URI hierarchy to be independent of the system organization, but there may be times when systems are initially created using the default URI hierarchy. When this is the case, the organization of the system can be changed, while maintaining the form of the URI hierarchy, by giving appropriate values to the ***uri*** attribute of select elements. Here are a couple examples of changes that can be made to the organization while maintaining the existing URIs:

- To turn a single moduleComponent (say "foo") into two moduleComponentss, the new moduleComponent that contains elements that used to be in "foo" should have a ***uri*** attribute whose value is: "../foo".

- To merge two moduleComponentss into a single moduleComponent ("foo" and "bar" into just "foo") the bindings that used to be in the "bar" module should have a ***uri*** attribute of "../bar/myEntryProint".

The URI attribute may also be used in order to create shorter URIs for some endpoints, where the subsystem or module component may not be present in the URI at all. For example, if a module component has a ***uri*** attribute of "." the module component name will not be present in the URI.

1.5.2 SCA Binding

The SCA binding element is defined by the following schema.

```
<binding.sca />
```

The SCA binding can be used for service interactions between clients and service contained in different SCA modules. The way in which this binding type is implemented is not defined by the SCA specification and it can be implemented in different ways by different SCA runtimes. The only requirement is that the required qualities of service must be implemented for the SCA binding type. The SCA binding type is ***not*** intended to be an interoperable binding type. For interoperability, an interoperable binding type such as the Web service binding should be used.

If an external service specifies an URI via its uri attribute, then this provides the default wire to a service provided by another module component. The value of the URI has to be as follows:

- <module-component-name>/<entrypoint-name>

1.5.2.1 Example SCA Binding

The following snippet shows the sca.module file for the MyValueModule file containing the entryPoint element for the MyValueService and an externalService element for the StockQuoteService. Both the entry point and the external service use an SCA binding. The target for the external service is left undefined in thjis binding and would have to be supplied by the subsystem in which this module is used.

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

   name="MyValueModule" >

   <entryPoint name="MyValueService">
         <interface.java interface="services.myvalue.MyValueService"/>
         <binding.sca/>
         …
   </entryPoint>

   …

   <externalService name="StockQuoteService">
         <interface.java interface="services.stockquote.StockQuoteService"/>
```

```
        <binding.sca/>
    </externalService>

</module>
```

1.5.3 Web Service Binding

The Web service binding element is defined by the following schema.

```
<binding.ws port="xs:anyURI"/>
```

The Web service binding element has a port attribute whose value is the URI of a WSDL port. The URI can have two forms:

- WSDL 1.1
  - <WSDL-namespace-URI>#wsdl.endpoint(<service-name>/<port-name>)
- WSDL 2.0
  - <WSDL-namespace-URI>#wsdl.endpoint(<service-name>/<endpoint-name>)

If an external service specifies an URI via its uri attribute, then this overrides the URI specified for the port/endpoint in the WSDL document that the port attribute points to.

For entry points the URI specified on the port/endpoint in the WSDL document that the port attribute points to is ignored.

Note: Future versions of the specification will provide other web service bindings that will not require the authoring of WSDL documents, e.g. web service bindings that support the WS-I Basic Profile, WS-I Security Profile, and so on.

1.5.3.1 Example Web Service Binding

The following snippet shows the sca.module file for the MyValueModule file containing the entryPoint element for the MyValueService and an externalService element for the StockQuoteService. Both the entry point and the external service use a Web service binding.

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

   name="MyValueModule" >

   <entryPoint name="MyValueService">
         <interface.java interface="services.myvalue.MyValueService"/>
         <binding.ws port="http://www.myvalue.org/MyValueService#
                     wsdl.endpoint(MyValueService/MyValueServiceSOAP)"/>
         …
   </entryPoint>
```

```
    …

    <externalService name="StockQuoteService">
          <interface.java interface="services.stockquote.StockQuoteService"/>
          <binding.ws port="http://www.stockquote.org/StockQuoteService#
                       wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
    </externalService>

</module>
```

1.6 Extension Model

The assembly model can be extended with support for new interface types, implementation types and binding types. The extension model is based on XML schema substitution groups and namespace federation.

The form of the element names for interfaces, implementations and bindings is chosen to enable extensibility of the schemas in a simple and easy to read format. So, the corresponding element names are interface.xxx, implementation.xxx and binding.xxx. "xxx" can take a name that relates to the type of interface, implementation or binding concerned. The SCA specification defines some basic types of interface, implementation and binding, but others can be defined as required, where support for these extra ones is available from the runtime.

Note: How to contribute SCA model extensions and their runtime function to an SCA runtime will be defined by a future version of the specification.

1.6.1 Defining an Interface Type

The following snippet shows the base definition for the ***interface*** element and ***Interface*** type contained in ***sca-core.xsd***; see appendix for complete schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema  xmlns="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://www.osoa.org/xmlns/sca/0.9"
         xmlns:sca="http://www.osoa.org/xmlns/sca/0.9">
   …

   <element name="interface" type="sca:Interface"/>
   <complexType name="Interface"/>

   …

</schema>
```

In the following snippet we show how the base definition is extended to support Java interfaces. The snippet shows the definition of the ***interface.java*** element and the ***JavaInterface*** type contained in ***sca-interface-java.xsd***.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema  xmlns="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://www.osoa.org/xmlns/sca/0.9"
         xmlns:sca="http://www.osoa.org/xmlns/sca/0.9">

   <element name="interface.java" type="sca:JavaInterface"
         substitutionGroup="sca:interface"/>
   <complexType name="JavaInterface">
         <complexContent>
               <extension base="sca:Interface">
                     <attribute name="interface" type="NCName" use="required"/>
               </extension>
         </complexContent>
   </complexType>
```

```
</schema>
```

1.6.2 Defining an Implementation Type

The following snippet shows the base definition for the ***implementation*** element and ***Implementation*** type contained in ***sca-core.xsd***; see appendix for complete schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema  xmlns="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://www.osoa.org/xmlns/sca/0.9"
         xmlns:sca="http://www.osoa.org/xmlns/sca/0.9">
   …

   <element name="implementation" type="sca:Implementation"/>
   <complexType name="Implementation"/>

   …

</schema>
```

In the following snippet we show how the base definition is extended to support Java implementation. The snippet shows the definition of the ***implementation.java*** element and the ***JavaImplementation*** type contained in ***sca-implementation-java.xsd***.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema  xmlns="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://www.osoa.org/xmlns/sca/0.9"
         xmlns:sca="http://www.osoa.org/xmlns/sca/0.9">

   <element name="implementation.java" type="sca:JavaImplementation"
                                 substitutionGroup="sca:implementation"/>
   <complexType name="JavaImplementation">
         <complexContent>
               <extension base="sca:Implementation">
                     <attribute name="class" type="NCName" use="required"/>
               </extension>
         </complexContent>
   </complexType>

</schema>
```

1.6.3 Defining a Binding Type

The following snippet shows the base definition for the ***binding*** element and ***Binding*** type contained in ***sca-core.xsd***; see appendix for complete schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema  xmlns="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://www.osoa.org/xmlns/sca/0.9"
         xmlns:sca="http://www.osoa.org/xmlns/sca/0.9">
   …

   <complexType name="ExternalProvider">
         <sequence>
```

```
            <element minOccurs="0" maxOccurs="1" ref="sca:binding"/>
            <element minOccurs="1" maxOccurs="unbounded" ref="sca:interface"/>
      </sequence>
      <attribute name="name" type="NCName" use="required"/>
   </complexType>

   …

   <element name="binding" type="sca:Binding"/>
   <complexType name="Binding"/>

   …

</schema>
```

In the following snippet we show how the base definition is extended to support Web service binding. The snippet shows the definition of the ***binding.java*** element and the ***WebServiceBinding*** type contained in ***sca-binding-webservice.xsd***.

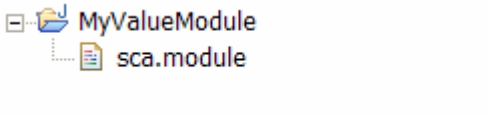
```
<?xml version="1.0" encoding="UTF-8"?>
<schema  xmlns="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://www.osoa.org/xmlns/sca/0.9"
         xmlns:sca="http://www.osoa.org/xmlns/sca/0.9">

   <element name="binding.ws" type="sca:WebServiceBinding"
         substitutionGroup="sca:binding"/>
   <complexType name="WebServiceBinding">
         <complexContent>
            <extension base="sca:Binding">
               <attribute name="port" type="anyURI" use="required"/>
            </extension>
         </complexContent>
   </complexType>
</schema>
```

2 Appendix 1

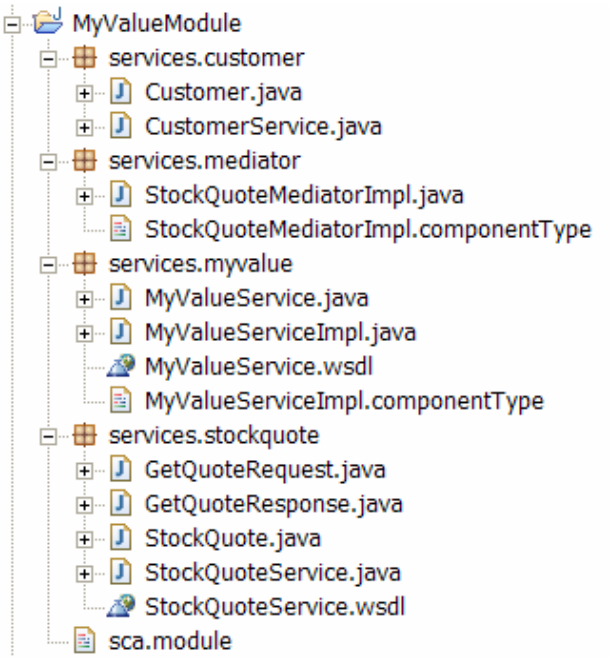
2.1 Packaging and Deployment

2.1.1 Module Packaging

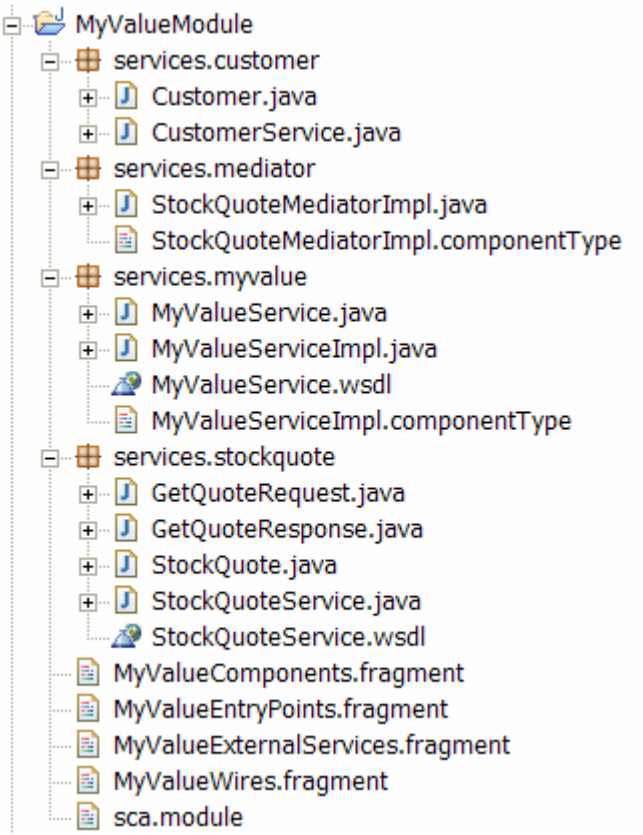
The physical realization of an SCA module can be a ***folder in a file system*** or an ***archive file*** (e.g. JAR) which is named like the module and which must contain one and only one ***sca.module*** file at its root. The following shows the MyValueModule just after creation in a file system.



Besides the sca.module file the module contains artifacts that define the implementations of components, and that define the bindings of external services and entry points. Examples of artifacts include Java interfaces, WSDL portTypes, XML schemas, Java implementation classes, BPEL implementation files, WSDL binding definitions, and so on. These artifacts can be contained in subfolders of the module, whereby programmers have the freedom to construct a folder structure that best fits the needs of their project. The following shows the complete MyValueModule folder file structure in a file system.



The following shows another variant of the complete MyValueModule folder file whereby here module fragments are used.



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

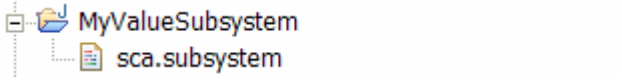
Java resources are referenced by module and component type files using fully qualified interface or class names. The module root is at the root of a class loader. The Java resources have to be loadable from the classpath of that class loader.

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

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

2.1.2 Subsystem Packaging

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



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

2.1.3 Deployment

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

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

The following shows the deployment of the MyValueModule, the CustomerModule and the MyValueSubsystem that configures the two modules. Modules and subsystems are deployed in archive form in this sample.

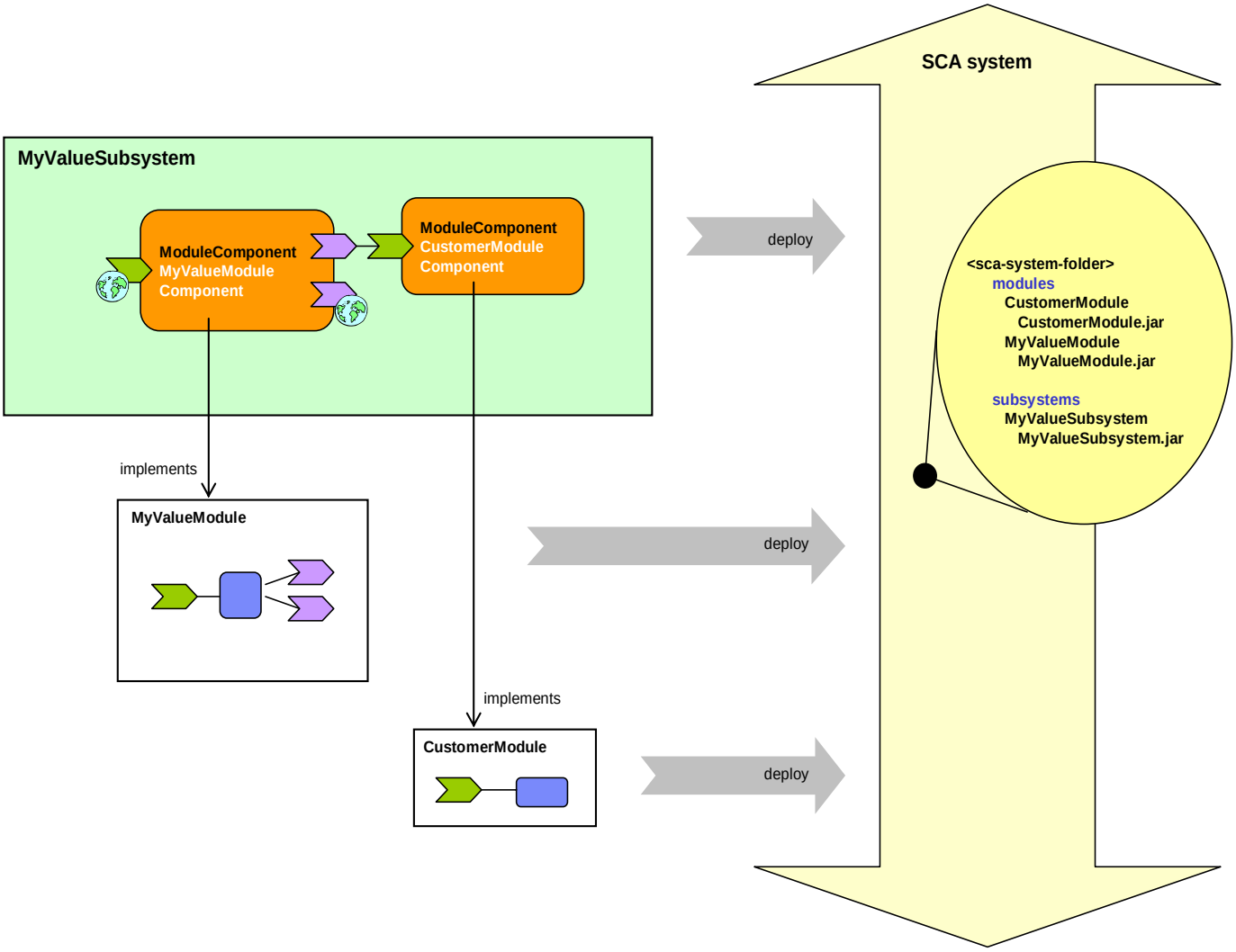


Figure 20: Subsystem deployment into a system

2.2 XML Schemas

2.2.1 sca.xsd

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

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

      <include schemaLocation="sca-interface-java.xsd"/>
      <include schemaLocation="sca-interface-wsdl.xsd"/>

      <include schemaLocation="sca-implementation-java.xsd"/>

      <include schemaLocation="sca-binding-webservice.xsd"/>
      <include schemaLocation="sca-binding-sca.xsd"/>

</schema>
```

2.2.2 sca-core.xsd

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

      <element name="componentType" type="sca:ComponentType"/>
      <complexType name="ComponentType">
            <sequence>
                  <element minOccurs="0" maxOccurs="unbounded" name="service"
                   type="sca:Service"/>
                  <element minOccurs="0" maxOccurs="unbounded" name="reference"
                   type="sca:Reference"/>
                  <element minOccurs="0" maxOccurs="unbounded" name="property"
                   type="sca:Property"/>
                  <any namespace="##other" processContents="lax" minOccurs="0"
                   maxOccurs="unbounded"/>
            </sequence>
            <anyAttribute namespace="##any" processContents="lax"/>
      </complexType>

      <complexType name="Service">
            <sequence>
                  <element minOccurs="1" maxOccurs="1" ref="sca:interface"/>
                <any namespace="##other" processContents="lax" minOccurs="0"
                   maxOccurs="unbounded"/>
            </sequence>
            <attribute name="name" type="NCName" use="required"/>
```

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

<element name="interface" type="sca:Interface"/>
<complexType name="Interface"/>

<complexType name="Reference">
      <sequence>
            <element minOccurs="1" maxOccurs="1" ref="sca:interface"/>
          <any namespace="##other" processContents="lax" minOccurs="0"
            maxOccurs="unbounded"/>
      </sequence>
      <attribute name="name" type="NCName" use="required"/>
      <attribute name="multiplicity" type="sca:Multiplicity" use="optional"
       default="1..1"/>
      <anyAttribute namespace="##any" processContents="lax"/>
</complexType>

<complexType name="Property">
      <sequence>
          <any namespace="##other" processContents="lax" minOccurs="0"
            maxOccurs="unbounded"/>
      </sequence>
      <attribute name="name" type="NCName" use="required"/>
      <attribute name="type" type="QName" use="required"/>
      <attribute name="many" type="boolean" default="false" use="optional"/>
      <attribute name="required" type="boolean" default="false"
       use="optional"/>
      <attribute name="default" type="string" use="optional"/>
      <anyAttribute namespace="##any" processContents="lax"/>
</complexType>


<element name="moduleFragment" type="sca:ModuleFragment"/>
<complexType name="ModuleFragment">
      <sequence>
            <element minOccurs="0" maxOccurs="unbounded" name="entryPoint"
             type="sca:EntryPoint"/>
            <element minOccurs="0" maxOccurs="unbounded" name="component"
             type="sca:Component"/>
            <element minOccurs="0" maxOccurs="unbounded" name="externalService"
             type="sca:ExternalService"/>
            <element minOccurs="0" maxOccurs="unbounded" name="wire"
             type="sca:ModuleWire"/>
            <any namespace="##other" processContents="lax" minOccurs="0"
             maxOccurs="unbounded"/>
      </sequence>
      <attribute name="name" type="NCName" use="required"/>
      <anyAttribute namespace="##any" processContents="lax"/>
</complexType>

<element name="module" type="sca:Module"/>
<complexType name="Module">
      <complexContent>
            <extension base="sca:ModuleFragment"/>
      </complexContent>
</complexType>

<complexType name="EntryPoint">
```

```
      <sequence>
            <element minOccurs="1" maxOccurs="1" ref="sca:interface"/>
            <element minOccurs="1" maxOccurs="unbounded" ref="sca:binding"/>
            <element minOccurs="1" maxOccurs="unbounded" name="reference"
             type="anyURI"/>
            <any namespace="##other" processContents="lax" minOccurs="0"
             maxOccurs="unbounded"/>
      </sequence>
      <attribute name="name" type="NCName" use="required"/>
      <attribute name="multiplicity" type="sca:Multiplicity" use="optional"
       default="1..1"/>
      <anyAttribute namespace="##any" processContents="lax"/>
</complexType>
<!-- a multiplicity 1..1 or 0..n sample
            <reference>StockQuoteComponent</reference>      - type must be URI
       a multiplicity 1..n or 0..n sample
            <reference>StockQuoteComponent1</reference>     - type must be URI
            <reference>StockQuoteComponent2</reference>
 -->

<element name="binding" type="sca:Binding"/>
<complexType name="Binding">
      <attribute name="uri" type="anyURI" use="optional"/>
</complexType>

<complexType name="Component">
      <sequence>
            <element minOccurs="1" maxOccurs="1" ref="sca:implementation"/>
            <element minOccurs="0" maxOccurs="1" name="properties"
             type="sca:PropertyValues"/>
            <element minOccurs="0" maxOccurs="1" name="references"
             type="sca:ReferenceValues"/>
            <any namespace="##other" processContents="lax" minOccurs="0"
             maxOccurs="unbounded"/>
      </sequence>
      <attribute name="name" type="NCName" use="required"/>
      <anyAttribute namespace="##any" processContents="lax"/>
</complexType>
<!-- a multiplicity 1..1 or 0..1 sample
            <references>
                  <v:stockQuote>
                        StockQuoteComponent
                  </v:stockquote>                     - type must be URI
            </references>
       a multiplicity 1..n or 0..n sample
            <references>
                  <v:stockQuote>StockQuoteComponent1</v:stockQuote>
                                                      - type must be URI
                  <v:stockQuote>StockQuoteComponent2</v:stockQuote>
            </references>
 -->

<element name="implementation" type="sca:Implementation"/>
<complexType name="Implementation"/>

<complexType name="PropertyValues">
      <sequence>
            <any namespace="##other" processContents="lax" minOccurs="0"
             maxOccurs="unbounded"/>
```

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

<complexType name="ReferenceValues">
      <sequence>
            <any namespace="##other" processContents="lax" minOccurs="0"
             maxOccurs="unbounded"/>
      </sequence>
      <anyAttribute namespace="##any" processContents="lax"/>
</complexType>


<complexType name="ExternalService">
      <sequence>
            <element minOccurs="1" maxOccurs="1" ref="sca:interface"/>
            <element minOccurs="0" maxOccurs="unbounded" ref="sca:binding"/>
            <any namespace="##other" processContents="lax" minOccurs="0"
             maxOccurs="unbounded"/>
      </sequence>
      <attribute name="name" type="NCName" use="required"/>
      <attribute name="overridable" type="sca:OverrideOptions" default="may"
       use="optional"/>
      <anyAttribute namespace="##any" processContents="lax"/>
</complexType>


<complexType name="ModuleWire">
      <sequence>
            <element minOccurs="1" maxOccurs="1" ref="sca:source.uri"/>
            <element minOccurs="1" maxOccurs="1" ref="sca:target.uri"/>
            <any namespace="##other" processContents="lax" minOccurs="0"
             maxOccurs="unbounded"/>
      </sequence>
      <anyAttribute namespace="##any" processContents="lax"/>
</complexType>

<element name="source" type="anyType"/>
<element name="target" type="anyType"/>

<element name="source.uri" type="anyURI" substitutionGroup="sca:source"/>
<element name="target.uri" type="anyURI" substitutionGroup="sca:target"/>

<element name="subsystem" type="sca:Subsystem"/>
<complexType name="Subsystem">
      <sequence>
            <element minOccurs="0" maxOccurs="unbounded" name="entryPoint"
             type="sca:EntryPoint"/>
            <element minOccurs="0" maxOccurs="unbounded" name="moduleComponent"
             type="sca:ModuleComponent"/>
            <element minOccurs="0" maxOccurs="unbounded" name="externalService"
             type="sca:ExternalService"/>
            <element minOccurs="0" maxOccurs="unbounded" name="wire"
             type="sca:SystemWire"/>
            <any namespace="##other" processContents="lax" minOccurs="0"
             maxOccurs="unbounded"/>
      </sequence>
      <attribute name="name" type="NCName" use="required"/>
      <attribute name="uri" type="anyURI" use="optional"/>
```

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

      <complexType name="ModuleComponent">
            <sequence>
                  <element minOccurs="0" maxOccurs="1" name="properties"
                   type="sca:PropertyValues"/>
                  <element minOccurs="0" maxOccurs="1" name="references"
                   type="sca:ReferenceValues"/>
                  <any namespace="##other" processContents="lax" minOccurs="0"
                   maxOccurs="unbounded"/>
            </sequence>
            <attribute name="name" type="NCName" use="required"/>
            <attribute name="module" type="NCName" use="required"/>
            <attribute name="uri" type="anyURI" use="optional"/>
            <anyAttribute namespace="##any" processContents="lax"/>
      </complexType>

      <complexType name="SystemWire">
            <sequence>
                  <element minOccurs="1" maxOccurs="1" ref="sca:source"/>
                  <element minOccurs="1" maxOccurs="1" ref="sca:target"/>
                  <any namespace="##other" processContents="lax" minOccurs="0"
                   maxOccurs="unbounded"/>
            </sequence>
      </complexType>

      <element name="source.epr" type="anyType" substitutionGroup="sca:source"/>
      <element name="target.epr" type="anyType" substitutionGroup="sca:target"/>

      <simpleType name="Multiplicity">
            <restriction base="string">
                  <enumeration value="0..1"/>
                  <enumeration value="1..1"/>
                  <enumeration value="0..n"/>
                  <enumeration value="1..n"/>
            </restriction>
      </simpleType>

      <simpleType name="OverrideOptions">
            <restriction base="string">
                  <enumeration value="no"/>
                  <enumeration value="may"/>
                  <enumeration value="must"/>
            </restriction>
      </simpleType>

</schema>
```

2.2.3 sca-interface-java.xsd

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

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

      <element name="interface.java" type="sca:JavaInterface"
       substitutionGroup="sca:interface"/>
      <complexType name="JavaInterface">
            <complexContent>
                  <extension base="sca:Interface">
                        <sequence>
                              <any namespace="##other" processContents="lax"
                               minOccurs="0" maxOccurs="unbounded"/>
                        </sequence>
                        <attribute name="interface" type="NCName" use="required"/>
                        <attribute name="callbackInterface" type="NCName"
                         use="optional"/>
                        <anyAttribute namespace="##any" processContents="lax"/>
                  </extension>
            </complexContent>
      </complexType>
</schema>
```

2.2.4 sca-interface-wsdl.xsd

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

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

      <element name="interface.wsdl" type="sca:WSDLPortType"
       substitutionGroup="sca:interface"/>
      <complexType name="WSDLPortType">
            <complexContent>
                  <extension base="sca:Interface">
                        <sequence>
                              <any namespace="##other" processContents="lax"
                               minOccurs="0" maxOccurs="unbounded"/>
                        </sequence>
                        <attribute name="interface" type="anyURI" use="required"/>
                        <attribute name="callbackInterface" type="anyURI"
                         use="optional"/>
                        <anyAttribute namespace="##any" processContents="lax"/>
                  </extension>
            </complexContent>
      </complexType>
</schema>
```

2.2.5 sca-implementation-java.xsd

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

```
            elementFormDefault="qualified">

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

      <element name="implementation.java" type="sca:JavaImplementation"
       substitutionGroup="sca:implementation"/>
      <complexType name="JavaImplementation">
            <complexContent>
                  <extension base="sca:Implementation">
                        <sequence>
                              <any namespace="##other" processContents="lax"
                               minOccurs="0" maxOccurs="unbounded"/>
                        </sequence>
                        <attribute name="class" type="NCName" use="required"/>
                        <anyAttribute namespace="##any" processContents="lax"/>
                  </extension>
            </complexContent>
      </complexType>
</schema>
```

2.2.6 sca-binding-sca .xsd

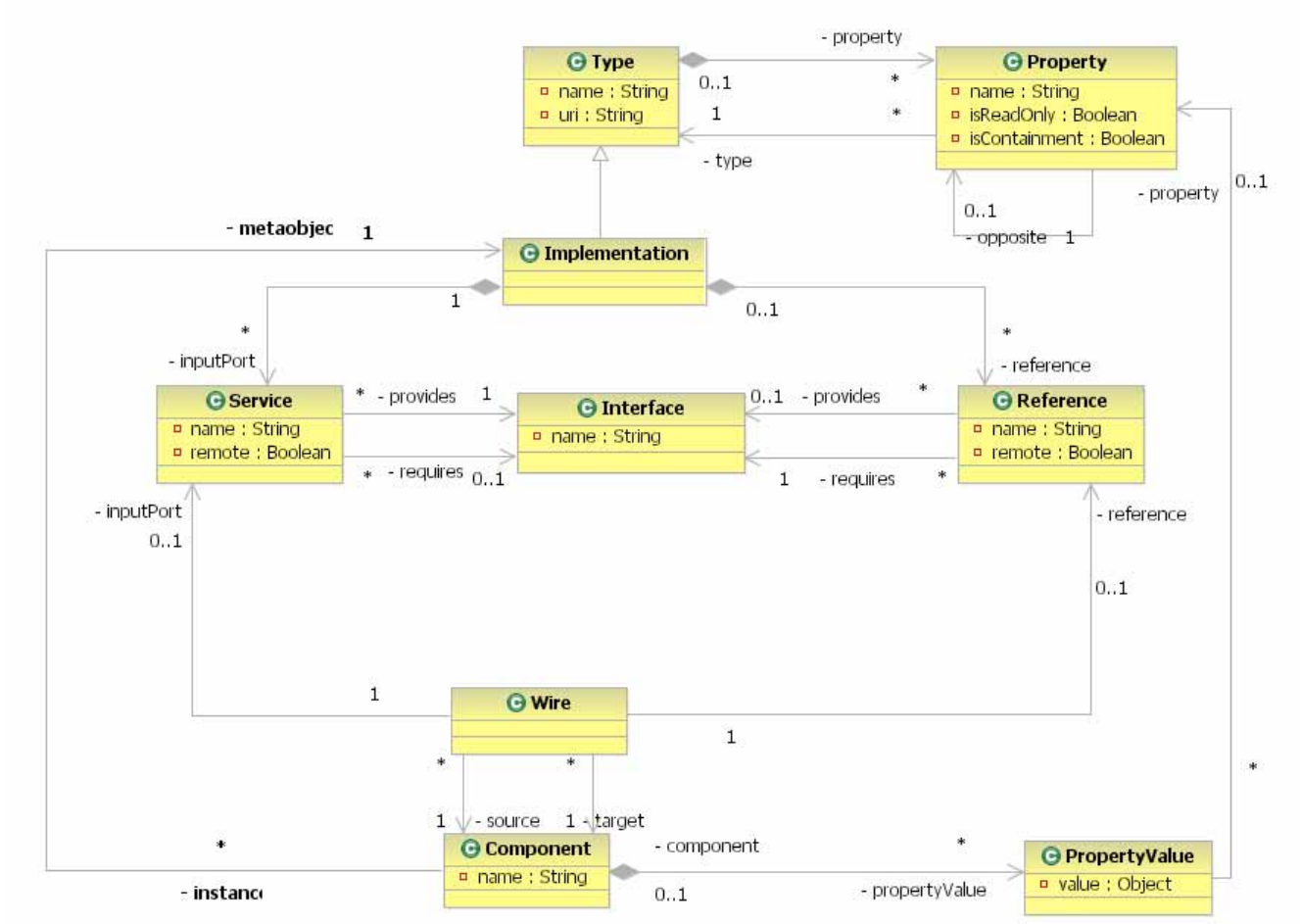
```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright BEA Systems Inc. and IBM Corporation 2005 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.osoa.org/xmlns/sca/0.9"
            xmlns:sca="http://www.osoa.org/xmlns/sca/0.9"
            elementFormDefault="qualified">

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

      <element name="binding.sca" type="sca:SCABinding"
       substitutionGroup="sca:binding"/>
      <complexType name="SCABinding">
            <complexContent>
                  <extension base="sca:Binding">
                        <sequence>
                              <any namespace="##other" processContents="lax"
                               minOccurs="0" maxOccurs="unbounded"/>
                        </sequence>
                        <anyAttribute namespace="##any" processContents="lax"/>
                  </extension>
            </complexContent>
      </complexType>
</schema>
```

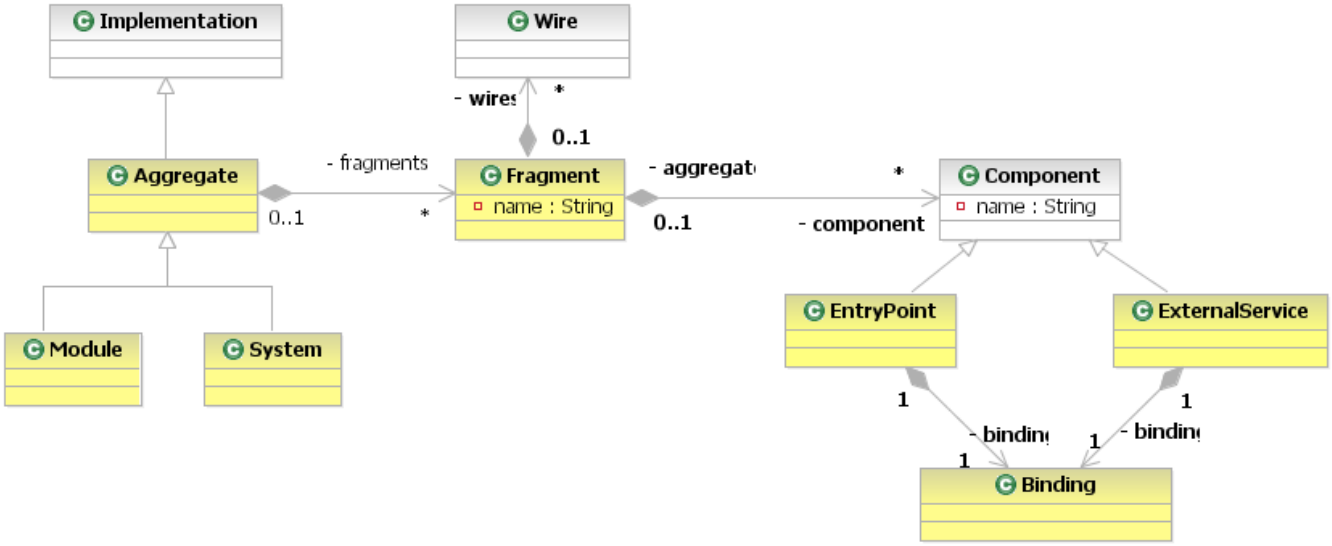
2.2.7 sca-binding-webservice.xsd

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

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

```
      <element name="binding.ws" type="sca:WebServiceBinding"
       substitutionGroup="sca:binding"/>
      <complexType name="WebServiceBinding">
            <complexContent>
                  <extension base="sca:Binding">
                        <sequence>
                              <any namespace="##other" processContents="lax"
                               minOccurs="0" maxOccurs="unbounded"/>
                        </sequence>
                        <attribute name="port" type="anyURI" use="required"/>
                        <anyAttribute namespace="##any" processContents="lax"/>
                  </extension>
            </complexContent>
      </complexType>
</schema>
```

2.3 UML Model




Implementation
Wire
- wires
*
0..1
Aggregate
- fragments
Fragment
name : String
- aggregat
*
Component
name : String
0..1
*
- component
EntryPoint
ExternalService
Module
System
1
1
- bindin
1
1
- bindin
Binding

2.4 SCA Concepts

2.4.1 Binding

Bindings are used by external service and entry points External services use bindings to describe the access mechanism used to call the service to which they are wired. Entry points use bindings to describe the access mechanism(s) that clients should use to call the service published by the entry point.

SCA supports multiple different types of bindings. Examples include ***SCA service, Web service, stateless session EJB, data base stored procedure, EIS service.*** SCA provides an extensibility mechanism by which an SCA runtime can add support for additional binding types.

2.4.2 Component

SCA components are configured instances of ***SCA implementations***, and provide and consume services. SCA allows many different implementation technologies for components such as Java, BPEL, C++. SCA defines an ***extensibility mechanism*** that allows you to introduce new implementation types. The current specification does not mandate the implementation technologies to be supported by an SCA run-time, vendors may choose to support the ones that are important for them. A single SCA implementation may be used by multiple Components, each with a different configuration.

The Component has a reference to an implementation of which it is an instance, a set of property values, and a set of service reference values. Property values define the values of the properties of the component as defined by the component's implementation. Service reference values define the services that resolve the service references of the component as defined by its implementation. These values can either be a particular service of a particular component, or an external service.

2.4.3 Entry Point

SCA entry points are used to declare the externally accessible services of a ***module***. Such a service can be a service provided by a component defined in the module, or by an external service defined in the module. The latter case allows the republication an external service with a new address and/or new bindings. The entry point can be thought of as a point at which messages from external clients enter the module.

An entry point may provide services ***as SCA remote services, as Web services, as stateless session EJB's, as EIS services, and so on***. Entry points use ***bindings*** to describe the way in which they publish the service provided by the module. SCA provides an ***extensibility mechanism*** that makes it possible to introduce new binding types for new types of entry points.

2.4.4 External Service

SCA external services represent remote services that are external to the SCA module that uses the remote service. These remote services can be accessed by components within a module like any service provided by an SCA component. External services can be used as the values of references when configuring Components.

An external service can be used to access a service such as: an SCA service provided by another SCA module, a Web service, a stateless session EJB, a data base stored procedure or an EIS service***,*** and so on. External services use ***bindings*** to describe the access to their services. SCA provides an ***extensibility mechanism*** that allows the introduction of new binding types to external services.

2.4.5 Implementation

An implementation is concept that is used to describe a piece of software technology such as a Java class, BPEL process, XSLT transform, or C++ class that is used to implement one or more services in a service-oriented application. Implementations also define points of variability including properties that can be set and settable references to other services. The points of variability are configured by

a component that uses the implementation. The specification refers to the configurable aspects of an implementation as its ***componentType***.

2.4.6 Interface

Interfaces define one or more business functions. These business functions are provided by Services and are used by components through References. Services are defined by the Interface they implement. SCA currently supports two interface type systems:

- Java interfaces
- WSDL portTypes

SCA also provides an extensibility mechanism by which an SCA runtime can add support for additional interface type systems.

Interfaces may be bi-***directional***. A bi-directional service has service operations which must be provided by each end of a service communication – this could be the case where a particular service requires a "callback" interface on the client, which is calls during the process of handing service requests from the client.

2.4.7 Module

An SCA module is the largest composition of tightly-coupled components that are developed and deployed together. It is the basic unit of loosely-coupled composition within an SCA System. An ***SCA Module*** is an assembly of Components, External services, Entry Points, and the Wires that interconnect them. Modules contribute service implementations to an ***SCA System***. A Module is used as the implementation of a Module Component.

A ***module*** has the following characteristics:

- It defines a boundary for Component visibility. Components may not be directly referenced outside of a module.
- Intra-module component communication is performed locally (except for services declared as Remotable); Inter-module component communication is performed remotely
- It defines a unit of deployment. Modules are used to contribute business logic artifacts to an SCA system.

2.4.8 Module Component

A module component is a configured instance of a module within a subsystem. The Modules used by a Module Component are said to ***implement*** the Module Component. In this use of Modules at the subsystem level, a Module is to a Component Type as a Module Component is to a Component. The Services provided by a Module component are defined by the Entry Points in the Module which implements the Module component. The References of a Module component are defined by the External Services of the Module which implements the Module component. The values of the service references of a module component may be services of other module components or external services of the system.

2.4.9 Module Fragment

A module fragment is a part of a module, provided to make team development of modules easier. A module fragment can contain any of the elements of a module. Module fragments are merged together at deployment time into a single logical module.

2.4.10 Property

Properties allow for the configuration of an implementation with externally set data values. The data value is provided by a Component or by a Module Component.

Each Property is defined by a property element in a componentType. Properties may be defined through annotations of implementations, where the implementation language permits, or through a componentType file. A Property can be either a simple data type or a complex data type. For complex data types, XML schema is the preferred technology for defining the data types.

2.4.11 Service

A service represents an addressable set of operations of an implementation, typed by an interface, that are designed to be either exposed for use by other implementations or exposed publicly for use elsewhere (eg public Web services for use by other organizations). The operations provided by a service are specified by an Interface, as are the operations required by the service client (if there is one). Services are implemented by implementations. A component may contain multiple services, when it is possible to address the services of a component separately.

2.4.11.1 Remotable Service

A Remotable Service is a service that is designed to be published remotely in a loosely-coupled SOA architecture. SCA services of SCA implementations can define implementations of industry-standard web services. Remotable services use pass-by-value semantics for parameters and returned results.
A service is remotable if it is defined by a WSDL port type or if it defined by a Java interface marked with the @Remotable annotation.

2.4.11.2 Local Service

Local services are services that are designed to be only used “locally” by other implementations that are deployed concurrently in a tightly-coupled architecture.

Such local usage can only be done from components that are in the same Module as the component which implements the service. Local services may rely on by-reference calling conventions, or may assume a very fine-grained interaction style that is incompatible with remote distribution. They may also use technology-specific data-types.

Currently a service is local only if it defined by a Java interface not marked with the @Remotable annotation.

2.4.12 Service Reference

A ***service reference*** (sometimes shortened to ***reference***) represents a dependency that an implementation has on a service that is supplied by some other implementation, where the service to be used is specified through configuration. In other words, a reference is a service that an implementation may call during the execution of its business function. Service references are typed by an interface and are owned by implementations. Which other implementation will resolve the reference is not “hard-coded” in the calling implementation – it is configured by the component which uses and configures the implementation.

2.4.13 Subsystem

A subsystem is used to group module components, entry points and external services of a system, plus the wires that interconnect them. The full configuration of an SCA system is represented by the combination of all the subsystems that are deployed into it.

2.4.14 System

An SCA System represents a set of Services providing an area of Business functionality that is controlled by a single organization. As an example, for the accounts department in a business, the SCA System might cover all finance-related functions, and it might contain a series of Modules dealing with specific areas of accounting, with one for Customer accounts, another dealing with Accounts Payable.

A system specifies the instantiation, configuration and connection of a set of modules. A configured instance of a module is called a Module Component. Systems, like Modules, also have Entry Points and External Services. Subsystems can also contain Wires which connect together the Module Components, Entry Points and External Services.

2.4.15 Wire

SCA wires connect ***service references*** to ***services***.

Within a Module, valid references are component references and entry points. Valid services are component services and external services.

Within a subsystem, valid references are moduleComponent references and susbsystem entry points. Valid services are moduleComponent services and subsystem external services. The sources and targets of the wires don't have to be part of the subsystem that contains the wire. The sources and targets can be defined by other subsystems. Targets can also be external to the SCA system.

2.5 Proposed Additional Features

The following are a list of features that have not been developed as fully as the features that are part of the main body of the specification. They may change substantially before being adopted by SCA or may not be adopted at all.

2.5.1 Extension Model

The current specification describes the way in which various aspects of SCA can be extended at the Assembly level. In particular, the form of SCDL XML files for extensions is described.

However, the specification does not define how runtime functionality for extensions would be added to the SCA runtime in order to implement the extensions in the runtime. This is clearly a gap in the specification. A future version of the SCA specification will describe a model for the SCA runtime and a model for adding extensions to the runtime.

2.5.2 Default Reference for External Services

Modules sometimes want to be able to expose external services that are intended to override some behavior that is present in the module. To make this scenario simpler to develop, the module's ***externalService*** element can optionally contain a ***defaultReference***, which may be wired to a service that is inside the module. If subsystem that deploys a module does not provide a value for that external service, then the service associated with the default reference is used. The runtime semantics of the default reference are exactly the same as if everything that is wired to the external service were instead wired to the target of the default reference.

This feature makes it possible to have default behavior without having to have special case code in the module that checks whether the external service had been set. This approach is also simpler than requiring that the external service always be set while also providing a module that exposes an entry point that should be used in order to get the default behavior. The schema for an external reference would change to be the following:

```
<externalService name="xs:NCName" override="sca:OverrideOptions"?>*
      <interface.interface-type/>
      <binding.binding-type/>
      <defaultReference>wire-target-URI</defaultReference>
</externalService>
```

3 Appendix 2: Policy, Security, Transactions, Reliable Messaging

This section is presented as an Appendix and not as a main part of the current specification. It is intended to provide a direction for the handling of aspects of Policy, Security, Transactions and Reliable Messaging within SCA. It is intended that this section will become part of the normative specification in a future updated version of the specification, modified to take account of community feedback on the material presented here. It is recognized that the treatment in this appendix is not complete, especially for Security. The eventual specification text will give a complete coverage of all the aspects described in this appendix.

3.1 Introduction

This document sets out the scope and model for what policy means in SCA terms, a general model and approach for SCA policy and concrete embodiments of specific domains of concern within the model – transactions, security and reliable message delivery. Policy is a much over-used term and means many things to many people. It is used in a wide and diverse range of topics such as governance and management, constraints and capabilities, internal configuration and behaviour and declarative programming. However, it is a widely accepted term for all of these, and rather than try and use different terms, this section will be specific when applying it to SCA.

To configure a running system with all the characteristics required to provide an overall level of service involves, in practice, a complex set of parameters. To simplify this task, the set of characteristics are divided into coherent sets of choices that represent a typical use case. These sets are known as ***Profiles***. A profile can be considered as an aggregation of policies from different domains of concern. The task of configuring the system is then broken down into a few of high-level decisions that reduce the need for many lower-level decisions, through the selection of the high-level profiles.

In SCA, a policy is a declaration of a specific set of behaviours within a particular domain of concern. Policy annotation refers to the way we augment the description of a model with declarations of particular policies. A specific transport protocol is just another quality of service, or policy, concern and so this spec proposes the use of a <Profile/> element as a replacement of the <Binding/> element, with transport protocol choice being one of the many QoS concerns expressed in a Profile.

In SCA, components within a module are all part of a common deployment package, and will share a common address space when deployed. When designing a component, or composing them within a module, there are no interoperability concerns – all components will be deployed in a homogeneous environment by a single role and interoperability between components is therefore assumed to be a given. The types of concern that may need to be declared or configured intra-module are internal, deployment independent concerns that will affect the internal behaviour of the module when deployed.

However, where EntryPoints or ExternalReferences are used, these are externalized parts of the SCA model that tie it to a particular deployment and runtime. They represent points at which concerns are external, where interoperability does become a concern. We differentiate between these two cases, internal and external, by grouping them into two categories – *Implementation* policy and *Interaction* policy.

It should be noted that SCA policy is a high-level abstraction of common qualify of service concerns and a means of indicating the selection of a set of detailed choices through simple

attributes. SCA policy always delegates to other, domain specific, metadata for describing the actual details of a particular quality of service choice.

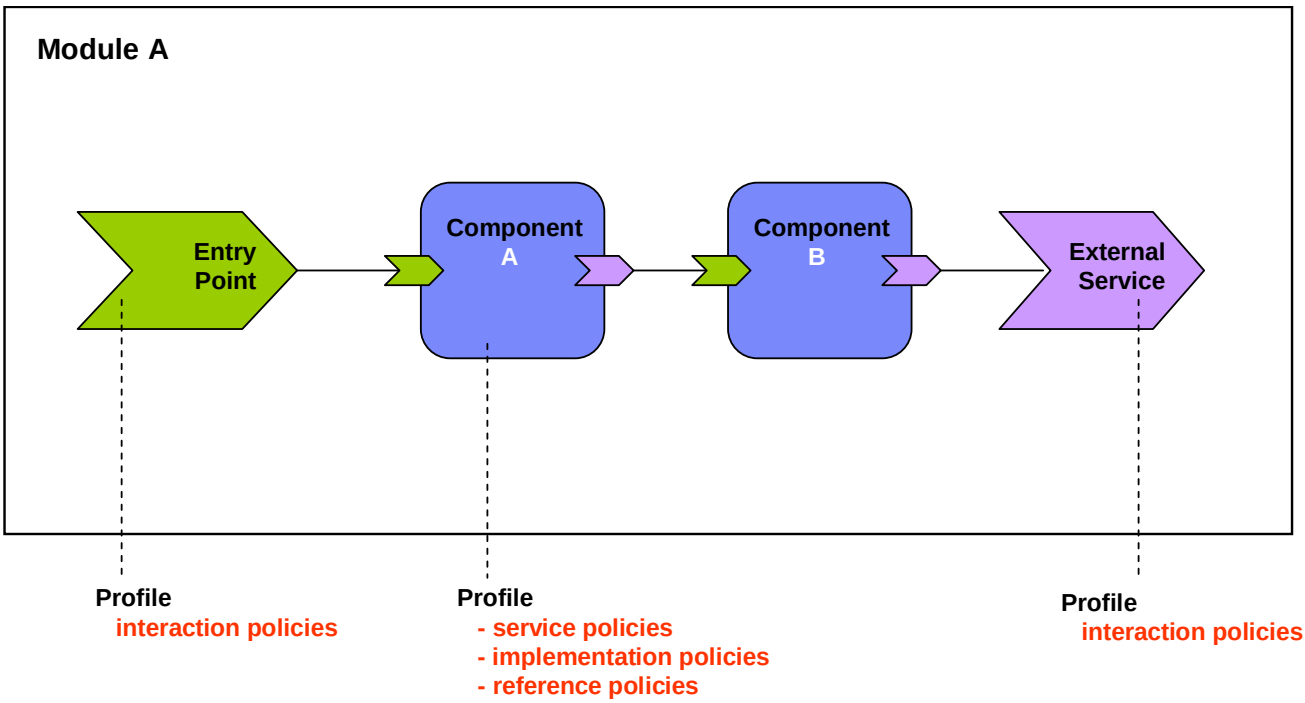


Figure 21: Profile attachment points within a Module

Implementation policies can only be attached to implementation profile elements, under component elements, and only denote internally relevant behaviours. By this we mean that the behaviours are local to the component they are attached to and only the container infrastructure that will host the deployed module will need to enforce the behaviour.

Interaction policies are attached to Profile elements on EntryPoints and ExternalServices and only denote externally relevant behaviours. This means that the behaviours may need to be enforced collaboratively with other parties using or used by the module, such as message encryption, context propagation and protocol exchanges.

The SCA policy annotation model consists of the following:

- Profile elements acting as an aggregation of a number of domain specific policy concerns. An example is <WSI_BP_1.1/>: a Profile to represent the WS-I Basic Profile [5].
- Policy attributes, defined as a part of the element schema, that are of a simple enumeration type, to act as a coarse grained control to indicate the selection of a specific policy within a particular domain. An example is :

  @messageProtection=”integrity|confidentiality|both”
- Policy elements that define, for a specific domain concern, the association between a particular abstraction of a quality of service for that domain, and a concrete XML file containing a representation of that quality of service in the relevant XML dialect. For interaction policies this is always a WS-Policy document. For implementation policies it may be some other native platform configuration file or other standardized form, e.g. XACML for access control.

The following diagram illustrates informally the relationship between these entities.

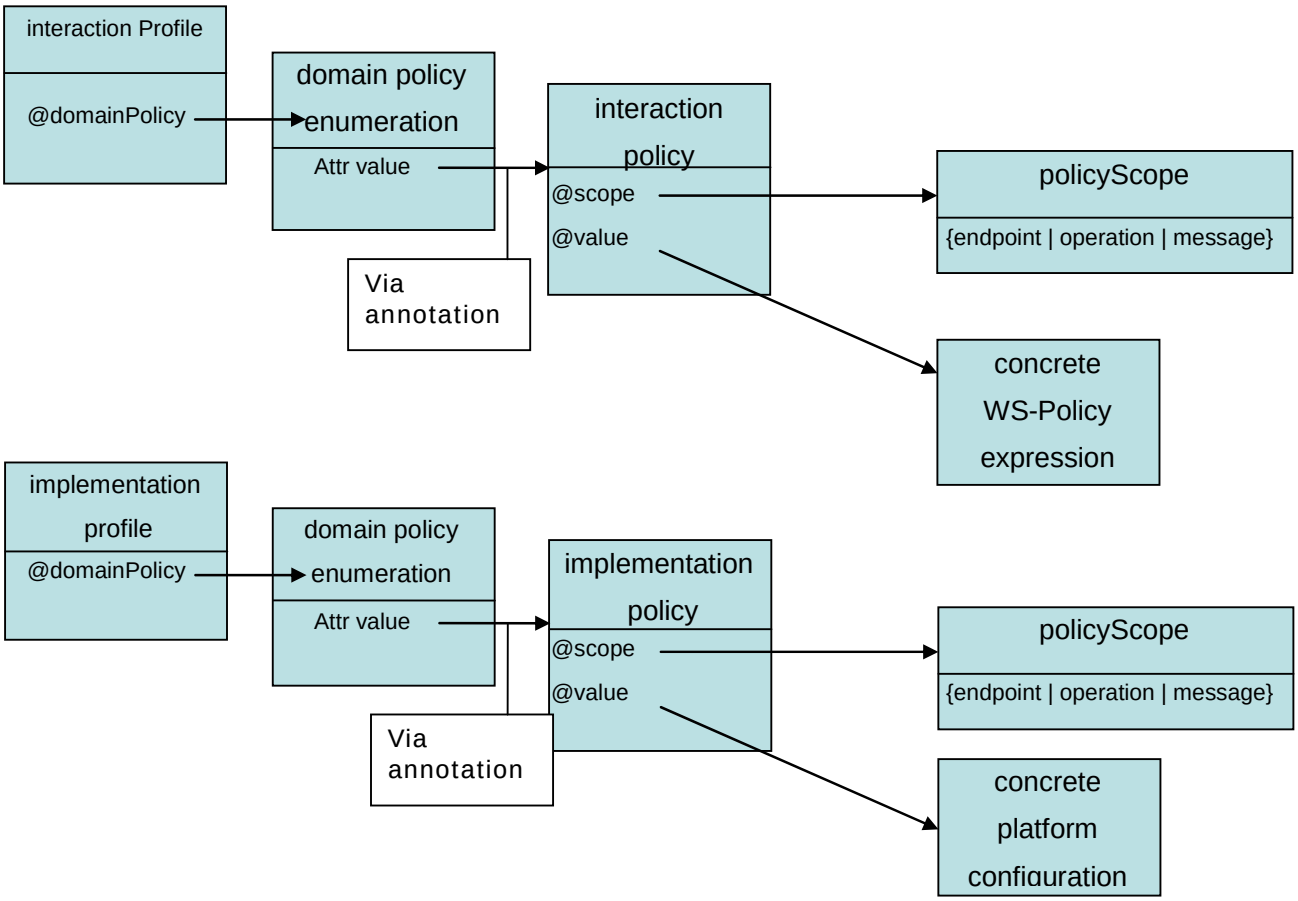


Figure 22: Entity relationships for profiles

An SCA policy represents a domain specific concern covering a complex set of characteristics that need to be manifest when the module is deployed. The simplest approach to configuration is to indicate a policy to use as a single entity that aggregates a whole set of particular choices around these characteristics – through the use of the corresponding Policy attribute. However, sometimes it is necessary to indicate variations in particular characteristic choices. Because of this, it is possible to extend the system to include other enumerations within the Policy attribute value range, and this method of extension allows the domain expert to associate other policies with the system as available choices. The method of extension will be described in more detail below.

Often, high level business objectives require a combination of policies to be set together to achieve that objective. Access, for example, requires a combination of authentication and authorization policies, but these can often be set independently. To aid in the composition of these policies, higher level policies can be defined to indicate a particular combination of policy annotations. These are known as System policies.

In summary, the meta model for a given SCA policy comprises of the following elements:-

1. Name – what is the name of this policy? Typically this will indicate the domain of concern.
2. Valid Attach Points – where can this policy be annotated in the SCA model?
3. Internal or External – does the behaviour the policy represents need to be externalized outside of the Module or can it be encapsulated within it?
4. Domain Characteristics – what individual characteristics does the policy aggregate together? These are encapsulated within the domain specific metadata for the policy.

5. Other Policies – if this is a System policy – which policies does it aggregate?

The document describes three SCA policies that are considered part of the base SCA specification – security, transactions and reliable delivery. The definition of custom policies is also described.

3.2 Scenario

To illustrate the model, let's consider a scenario where a TravelService component is being deployed in a module with an external service, with certain qualities of service regarding its own behaviour, and with certain required qualities of service from the services it depends on – FlightService, RentalCarService and HotelService.

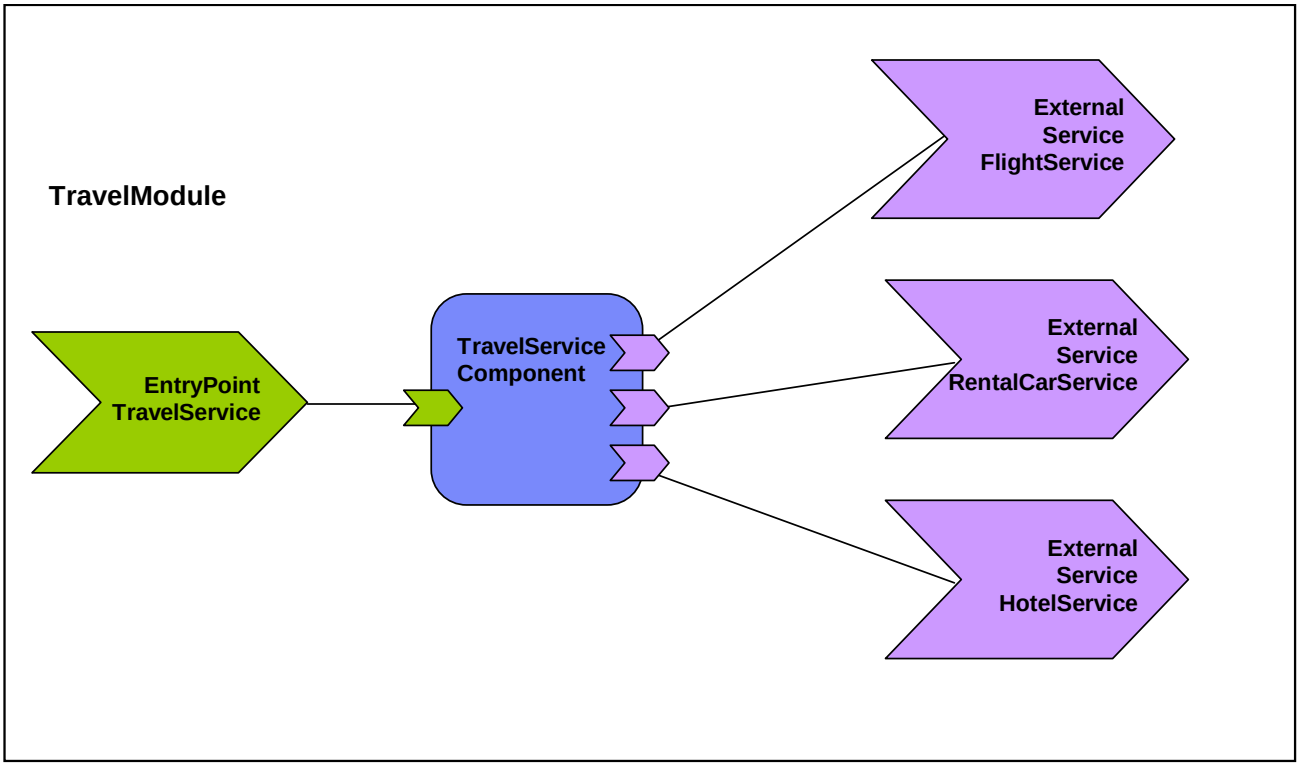


Figure 23: TravelModule module diagram

This example demonstrates how the SCDL is annotated at the appropriate points to indicate the quality of service requirements through SCA policy. The basic SCDL for this example looks like this:-

```
<?xml version="1.0" encoding="UTF-8"?>
<sca:module name="TravelModule"
         xmlns:sca="http://www.osoa.org/xmlns/sca/0.9"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.osoa.org/xmlns/sca/0.9 sca.xsd ">

   <sca:entryPoint name="TravelService">
         <sca:interface.wsdl portType="http://www.osoa.org" />
         <sca:binding />
         <sca:reference />
   </sca:entryPoint>
   <sca:component name="TravelServiceComponent">
         <sca:implementation />
   </sca:component>
```

```
    <sca:externalService name="FlightService">
          <sca:interface />
          <sca:binding />
    </sca:externalService>
    <sca:externalService name="RentalCarService">
          <sca:interface />
          <sca:binding />
    </sca:externalService>
    <sca:externalService name="HotelService">
          <sca:interface />
          <sca:binding />
    </sca:externalService>
</sca:module>
```

And the TravelServiceComponent has the following ComponentType:

```
<?xml version="1.0" encoding="UTF-8"?>
<sca:componentType xmlns:sca="http://www.osoa.org/xmlns/sca/0.9"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.osoa.org/xmlns/sca/0.9 sca.xsd ">

    <sca:service name="TravelService">
          <sca:interface />
    </sca:service>
    <sca:reference name="FlightService">
          <sca:interface />
    </sca:reference>
    <sca:reference name="CarRentalService">
          <sca:interface />
    </sca:reference>
    <sca:reference name="HotelService">
          <sca:interface />
    </sca:reference>
</sca:componentType>
```

3.3 Interaction Policies

Interaction policies are used to configure externally influencing behaviours for External Services and Entry Points, through Profile elements. The annotation of policies on these elements are a means of constraining and influencing the possible endpoints that might be wired to these elements when the Module is deployed, and factor into compatibility matching alongside interface type checking.

The absence of Policy annotations on External References or Entry Points means there are no constraints specified (beyond the interface type) regarding the wiring of endpoints to these elements. Conversely, the more Policy annotations added, the more specific a particular endpoint must be.

3.3.1 Entry Points

The use of Interaction policy on Entry Points is a way of defining what policies will be enforced at that Entry Point and, consequently, what quality of service clients of the Entry Point should expect.

In our example, we need to annotate the SCDL to add an appropriate Profile to represent the quality of service we want our Travel Service to offer. Let's assume that we want it to simply be a web service conformant with the WS-I Basic Security Profile 1.0 [6], and all that implies. This would cover the basic SOAP over HTTP interoperability, WSDL 1.1 and XML usage, as defined in the WS-I Basic Profile 1.1, and additionally the usage of WS-Security to protect the message

exchanges. To do this, we would simply switch the default <sca:binding/> to use the standard <sca:WSI_BSP_1.0/> Profile:

```
<sca:entryPoint name="TravelService">
      <sca:interface.wsdl portType="http://www.osoa.org" />
      <sca:WSI_BSP_1.0 />
      <sca:reference />
</sca:entryPoint>
```

If you look at the schema for this profile (given in WSI_BP_1.1 section), you will see that it contains a number of policy attributes related to security. They do, however, have default values, and for a plain, compliant, set of policy choices it is possible to just leave these attributes unset.

Where clients of an entry point are not a part of the SCA system the entry point is in, or not an SCA-enabled client, the policies associated with an Entry Point are externalized from the deployed SCA model and made available to those clients in an interoperable form. This is achieved by representing the external policy concerns as WS-Policy expressions, and appropriately annotated WSDL, to describe the Entry Point.

An Interaction policy definition describes the interface ***Scope*** (whether it applies an endpoint an operation or a message) and has a pointer to the WS-Policy expression that describes the behaviour. The metadata is used to facilitate the generation of WSDL annotated with the relevant WS-Policies. For a given Profile element, the Interaction policies on it are represented in WSDL as a set of WS-Policy annotations throughout a single WSDL binding tree for the various endpoint, operation and message specific attachments. This detailed information is hidden from the view of the SCA programmer and assembler, and appears as an annotation to the schema of the definition of the policy attributes types.

For example, the messageProtection policy attribute that is included in the definition of the WSI_BSP_1.0 profile is of type messageProtectionPolicy, which is an enumeration with annotations as follows:-

```
<simpleType name="messageProtectionPolicy">

  <restriction base="string">
            <enumeration  value="integrity" >
                  <annotation>
                        <appinfo>
                              <sca:messageProtection
                                    interactionScope="sca:messageScope"

value="sca:policy/security/messageProtection/default.xml" />
                        </appinfo>
                  </annotation>
            </enumeration>
            <enumeration value="confidentiality">
                  <annotation>
                        <appinfo>
                              <sca:messageProtection.medium
                                    interactionScope="sca:messageScope"

value="sca:policy/security/messageProtection/medium.xml" />
                        </appinfo>
                  </annotation>
            </enumeration>
```

```
                <enumeration value="both">
                      <annotation>
                            <appinfo>
                                  <sca:messageProtection.strong
                                        interactionScope="sca:messageScope"
   value="sca:policy/security/messageProtection/strong.xml" />
                            </appinfo>
                      </annotation>
                </enumeration>
          </restriction>
   </simpleType>
```

In this example, the base defaults of the WSI_BSP_1.0 profile are used, which include messageProtection="integrity". This implies, through the annotation of the "integrity" value, that the <sca:messageProtection /> policy element has been chosen. This policy element has fixed value attributes and defaults. It's "scope" attribute is fixed to be "sca:message" and its default "value" for which WS-Policy file to associate is "sca:policy/security/messageProtection/default.xml".

For example purposes only, the default.xml may look something like this:

```
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
   xmlns:sp=http://schemas.xmlsoap.org/ws/2005/02/securitypolicy
   wsp:Id="sca:policy/security/messageProtection/default.xml">
   <wsp:ExactlyOne>
         <wsp:All>
               <sp:SignedParts>
                     <sp:Body/>
                     <sp:Header
               Namespace="http://schemas.xmlsoap.org/ws/2004/08/addressing"/>
               </sp:SignedParts>
               <sp:EncryptedParts>
                     <sp:Body/>
               </sp:EncryptedParts>
         </wsp:All>
   </wsp:ExactlyOne>
</wsp:Policy>
```

As can be seen, default.xml contains a WS-Policy expression with WS-SecurityPolicy assertions to define the specific security characteristics for message protection, including which parts of the message should be signed and encrypted.

An alternative to SCA Interaction policy annotations is for the profile element to specify a concrete WSDL, with WS-Policy attachments, explicitly. The WSDL/WS-Policy is then used directly. This is useful where the deployer wishes to use a pre-defined set of WSDL/WS-Policy documents rather than synthesize them through SCA. We shall look at this approach in more detail in the following section, in the case of External References.

The WSDL and WS-Policy for the Entry Point is the external metadata describing the contract of an endpoint implementing this Entry Point. It can either be exported statically from the system via tooling, or it can be made available to requestors at runtime through WS-MetadataExchange requests to the endpoint. From our above example, the WSDL that would be generated for our TravelService Entry Point would look like this:-

WSDL + WS-Policy URI references to go in here…quick sketch:-

```
<wsdl:definitionsl>
<wsdl:portType name="TravelService"/>
<wsdl:binding type="TravelService">
    <wsdl:operation>
        <wsdl:input>
            <wsp:policyReference URI="sca:policy/security/messageProection/default.xml"/>
...
```

3.3.2 External Services

The use of Interaction policy on External Services is a way of defining what policies a suitable provider of the External Service should implement. As with the Interaction policies on Entry Points, there is a direct referential representation of these SCA policies as WS-Policy expressions and the corresponding WSDL for the portType of the interface. These WS-Policy expressions represent the Effective Policies for the various policy subjects (endpoint/operation/message) and they are used in the process of policy matching with prospective service providers.

In the example, assume that the interactions with the FlightService need to be reliable and secure. A profile must be specified that offers both these domains of concern. A profile that does this is the Reliable Access and Messaging Profile – IBM_RAMP_1.0. To indicate this in the SCDL, it is specified in profile element:

```
<sca:externalService name="FlightService">
      <sca:interface />
      <sca:IBM_RAMP_1.0/>
</sca:externalService>
```

The schema for this profile (see IBM_RAMP_1.0 section) extends the WSI_BSP_1.0 profile with the addition of a “reliableDelivery” policy attribute, giving both security and reliable delivery control.

Continuing the example, same profile can be used for the RentalCarService, but there may be a need for a greater level of security with the rental car service, and the standard defaults are not enough. In this case, the same profile element is used, but override values are provided for the security policy attributes:

```
<sca:externalService name="RentalCarService">
      <sca:interface />
      <sca:IBM_RAMP_1.0 authentication="cert"
messageProtection="confidentiality" />
</sca:externalService>
```

These values indicate the use of certificate based authentication (as opposed to the default of Basic) and encryption of the messages (as opposed to just signing them). The annotated schema for these values refers to different instances of the corresponding policy elements, and different WS-Policy expressions. More detail of this is given in the Policies section.

Once again, an alternative approach may be to directly reference a concrete WSDL with WS-Policy annotations. For example, assuming that there is a concrete HotelService provider which will always be used, the WSDL provided by this HotelService can be used:

```
<sca:externalService name="HotelService">
      <sca:interface />
      <sca:binding.ws wsdlBinding="hts:HotelServiceSOAP" />
</sca:externalService>
```

3.3.3 Policy Meta Model

The Profile attachment diagram illustrated the hierarchical structure of the policy meta-model for both interaction and implementation policies. The following snippets show the schema for the constituent parts, and how these elements combine.

Firstly, the base abstract types for policies – used to define domain specific policy elements.

```
<element name="interactionPolicy" type="sca:InteractionPolicy"
         abstract="true"/>
<complexType name="InteractionPolicy">
      <attribute name="scope" type="sca:interactionScope"/>
      <attribute name="value" type="anyURI"/>
</complexType>

<element name="implementationPolicy" type="sca:ImplementationPolicy"
         abstract="true"/>
<complexType name="ImplementationPolicy">
      <attribute name="scope" type="sca:implementationScope"/>
      <attribute name="value" type="anyURI"/>
</complexType>

<simpleType name="interactionScope">
      <restriction base="anyURI">
            <enumeration value="sca:serviceScope"/>
            <enumeration value="sca:endpointScope"/>
            <enumeration value="sca:operationScope"/>
            <enumeration value="sca:messageScope"/>
      </restriction>
</simpleType>
<simpleType name="implementationScope">
      <restriction base="anyURI">
            <enumeration value="sca:endpointScope"/>
            <enumeration value="sca:operationScope"/>
      </restriction>
</simpleType>
```

There are two base policy types – InteractionPolicy and ImplementationPolicy – that have two attributes, scope and value. Two elements are also defined, <interactionPolicy/> and <implementationPolicy/>, of the respective types to provide an abstract element and substitution group for domain extensions to be built upon. For a given SCA policy, a domain expert may extend the appropriate type and create an element with the required values of these attributes fixed, and make the element a part of a substitutionGroup to collect the domain specific policies together. This is shown in more detail in the Policies section.

For interaction policies, the value attribute is a URI that should reference a concrete WS-Policy expression that represents the SCA policy. For implementation policies, this value may point to some platform specific metadata, or other standardized form for that domain.

The possible values of the scope of an interaction policy are restricted to either “service”, “endpoint”, “operation” or “message”. These are the canonical policy subjects defined by the WS-

PolicyAttachment specification and used to determine the effective policy of various Web service attachment models, with these points representing an entire WSDL 1.1 web service, a specific endpoint of a web service embodied by a port, a particular message exchange pattern embodied by a Web service operation, and individual messages within an exchange. For more information, the reader should refer to this specification. For the purposes of SCA, these are hidden from the end user of SCA, and are required to enable the synthesis of WSDL and WS-Policy attachments from SCA policy.

When a concrete, domain-specific policy extends this abstract policy type, it should fix the relevant value for it. This scope indicates where the associated WS-Policy expression for the policy applies. This can enable the WS-Policy expression to be processed as part of broader web service processing such as the annotation of WSDL with the WS-Policy expressions, and the calculation of effective policy between source and target endpoints.

An example of two concrete interaction policies within the same substitutionGroup is as follows:

```
<element name="authentication" type="sca:security.Authentication"
      substitutionGroup="sca:interactionPolicy" />
<complexType name="security.Authentication">
      <complexContent>
            <restriction base="sca:InteractionPolicy">
                  <attribute name="scope" type="sca:interactionScope"
                              fixed="sca:endpointScope" />
                  <attribute name="value" type="anyURI"
                  default="sca:policy/security/authentication/default.xml" />
            </restriction>
      </complexContent>
</complexType>

<element name="authentication.basic"
      type="sca:security.Authentication.basic"
      substitutionGroup="sca:authentication" />
<complexType name="security.Authentication.basic">
      <complexContent>
            <restriction base="sca:security.Authentication">
                  <attribute name="value" type="anyURI"
                        fixed="sca:policy/security/authentication/basic.xml" />
            </restriction>
      </complexContent>
</complexType>
```

Here, a concrete SCA policy element, authentication, has been declared as a type that extends sca:InteractionPolicy and is in the substitution group of sca:interactionPolicy. The type sca:Authentication, that restricts the abstract type, fixes the value of the scope attribute to indicate this policy applies to endpoints, and defaults the value attribute to reference the WS-Policy expression contained at sca:policy/security/authentication/default.xml. The second policy element, sca:Authentication.basic, further restricts this base authentication policy element to a different WS-Policy expression – basic.xml.

In order to make interaction policies a part of the profile definition, they must have a policy attribute to index them. For the authentication policy element, the following @authentication attribute is defined, for use on profile elements.

```
<simpleType name="authenticationPolicy">
      <restriction base="string">
```

```
            <enumeration value="basic">
                  <annotation>
                        <appinfo>
                              <sca:authentication.basic
                                    interactionScope="sca:messageScope"
                        value="sca:policy/security/messageProtection/basic.xml" />

                        </appinfo>
                  </annotation>
            </enumeration>
            <enumeration value="cert">
                  <annotation>
                        <appinfo>
                              <sca:authentication.cert
                                    interactionScope="sca:messageScope"
                        value="sca:policy/security/messageProtection/cert.xml" />
                        </appinfo>
                  </annotation>
            </enumeration>
            <enumeration value="ltpa">
                  <annotation>
                        <appinfo>
                              <sca:authentication.ltpa
                                    interactionScope="sca:messageScope"
                        value="sca:policy/security/messageProtection/ltpa.xml" />
                        </appinfo>
                  </annotation>
            </enumeration>
            <enumeration value="digest">
                  <annotation>
                        <appinfo>
                              <sca:authentication.digest
                                    interactionScope="sca:messageScope"
                        value="sca:policy/security/messageProtection/digest.xml" />
                        </appinfo>
                  </annotation>
            </enumeration>
      </restriction>
</simpleType>
```

Finally, a profile that uses this policy would then define it's usage with a default value, so that a user of the profile is not required to explicitly set it in order to take advantage of it.

```
<element name="WSI_BSP_1.0" type="sca:WSBasicSecurityProfile"
                  substitutionGroup="sca:binding"/>
<complexType name="WSBasicSecurityProfile">
      <complexContent>
            <extension base="sca:WSBasicProfile">
                  <attribute name="messageProtection"
                              type="sca:messageProtectionPolicy"
                              default="integrity"/>
                  <attribute name="authentication"
                              type="sca:authenticationPolicy"
                              default="basic"/>
            </extension>
      </complexContent>
```

```
   </complexType>
```

Although the current approach is to use schema <annotation> to associate a policy attribute value with a policy element, a best practice should be that the name of the policy attribute mirrors the base policy element name, and the enumerated values reflect the extension of that base – e.g. @authentication=”basic” implies the policy element <authentication.basic/>.

3.4 Implementation Policies

Implementation policies have the same meta-model as interaction policies, described above. However, in order to use them, the SCA profile element is applied to componentTypes, as a means of aggregating implementation policies together for a component.

In our example scenario above, the TravelService is exposed with a RAMP 1.0 compliant entry point and this includes policy statements about authentication. Assume that policy about who should be authorized to use the TravelService must be specified, based on their authenticating credentials. This is done by applying an implementationProfile to the componentType for the TravelService component, that includes a policy attribute for identity.

```
<sca:componentType xmlns:sca="http://http://www.osoa.org/xmlns/sca/0.9"
                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://http://www.osoa.org/xmlns/sca/0.9 sca.xsd ">

   <sca:service name="TravelService">
         <sca:interface />
   </sca:service>
   <sca:reference name="FlightService">
         <sca:interface />
   </sca:reference>
   <sca:reference name="CarRentalService">
         <sca:interface />
   </sca:reference>
   <sca:reference name="HotelService">
         <sca:interface />
   </sca:reference>
   <sca:implementationProfile.enterprise identity="MyTravellerRoles" />
</sca:componentType>
```

Here, a concrete SCA policy element, identity.MyTravellerRoles has been implied by the setting of this attribute. This element points to the XML file to describe the users that are a member of the MyTravellerRoles. This file may be an XACML file, or a proprietary system format – whatever the target deployment systems security manager will use to define roles.

3.5 System Policies

This will describe how a combination of interaction and implementation policy profiles can be specified on a moduleComponent.

3.6 Policies

The following section lists the standard SCA policy elements and attributes for the security, transaction and reliable delivery domains. The format of the entries is as follows:-

Type of Policy

- Base policy name
    - Specific policy instances

3.6.1 Security

The intent of the SCA security model is to define a policy language that specifies security constraints at a level of abstraction high enough to reflect business objectives. The SCA security policy should be agnostic to any security mechanism and to a particular security protocol.

In addition to the base profile model presented in this document, the SCA security model allows for business logic developers to declaratively state the security requirements of the application. For example, developers are aware of the use of sensitive business data by the application, and can express the desire for message protection in the business interface.

We define the following policies for security to be a base part of SCA.

Interaction Policies

- messageProtection – Describes the need to digitally sign or encrypt a message in order to protect it from fraudulent activity. Digital signatures prevent modification of messages in-flight. Message encryption conceals the contents of the message from everyone but the sender and the intended receiver.
- authentication – Describes the mechanism by which a client provide proof of identity. For example, basic authentication where the client provides a userid and a password.

Implementation Policies

- identity – Describes a mechanism that allows a business operation to execute with a specified set of permissions defined by a role, aka RunAs.
- permission – Describes a role based mechanism whereby clients of a service must be in a particular role in order to execute operations on the target service.

3.6.2 Transactions

SCA recognizes that the presence or absence of ACID transaction coordination has a direct effect on how business logic is coded. In the absence of ACID transactions, developers must provide logic that compensates for failures. In the presence of ACID transactions, the underlying infrastructure is responsible for ensuring the consistency of all interactions. SCA supports both programming styles. SCA provides declarative mechanisms for describing the transactional environment required by the business logic. The rest of this section describes each of the declarative aspects in detail.

The SCA Client and Implementation specification will define an annotation for this declaration.

Interaction Policies

- joinsTransaction - entryPoints and externalServices need the ability to specify how they will handle a propagated transaction context.

Implementation Policies

The following is a brief list of each transactional aspect that is supported by SCA. More details can be found in the following sections.

- transaction – Service component's transactional environment.
- joinsTransaction – Describes how the SCA container handles propagated transaction context.
- suspendClientTransaction – Determines whether the transaction should be suspended when invoking operations on a particular service reference.
- invokeAsync – Describes how OneWay invocations are transacted.

3.6.2.1 Transaction Environment

The ***transaction*** policy describes the transactional environment required by a service component implementation, and is therefore a contract between a service component implementation and its hosting container. SCA provides transaction environments that are managed by the SCA container in order to remove the burden of coding transaction APIs directly into the business logic. The valid values for a transaction policy are:

- **global** – There must be an atomic transaction in order to run this component. The SCA runtime must ensure that a global transaction is present before dispatching any method on the component. The SCA runtime MAY use a transaction propagated from a client or MAY begin and complete a new transaction. See the joinsTransaction declaration below for more details.
- **local** – The component cannot tolerate running as part of an atomic transaction, and will therefore run within a local transaction containment (or with no transaction containment at all if the hosting environment does not support transaction management). A transaction that is propagated to the hosting SCA runtime MUST not be joined by the hosting runtime on behalf of this component. Local transactions are not propagated outbound across remotable interfaces. When interacting with a resource manager in the local transaction environment, it is possible for the application to control the transaction boundaries. This aspect of local transactions is described below in the section called "Local Transactions".

- **any** (default) - The component is agnostic to the presence or absence of a specific transactional environment. The SCA runtime will dispatch methods on the component in a global transaction if a global transaction context has been propagated from the client or else the SCA runtime will establish a local transaction environment.

Local Transactions

It is not enough to simply declare that a piece of business logic runs with no global transaction context. The absence of a global transaction cannot mean that the business logic is restricted from accessing resource managers which are capable of supporting atomic transactions. The business logic developer still needs to know the expected semantic of making one or more calls to one or more resource managers, and needs to know when and/or how those interactions will be committed, e.g. whether or not to expect a binary outcome. The term *local transaction containment* is used to describe the environment where there is no global transaction. By default, the boundaries of an LTC are scoped to a remotable service provider method.

The two most common patterns for components using resource managers outside a global transaction are:

- The application desires each interaction with a resource manager to commit after every interaction.
- The application desires each interaction with a resource manager to be part of an extended local transaction that is committed at the end of the method.

While an application may use Java interfaces provided by the resource adapter to explicitly demarcate resource manager local transactions (RMLT), this is a generally undesirable burden on applications which typically prefer all transaction considerations to be managed by the container. In addition, once an application codes to a Java local transaction interface, it may never be redeployed with a different transaction attribute since local transaction interfaces may not be used in the presence of a global transaction. It is therefore desirable for an application to be able to choose between the above two local transaction semantics via a declaration on the implementation.

A service component declares which of the two local transaction patterns it wishes to use via the **localTranResolver** declaration. The choice of these 2 distinct local transaction semantics needs to be expressible at a service component implementation level and only applies when the transaction environment value is local. The valid values for **localTranResolver** are:

- container (default) – The container wraps interactions with each resource manager in a resource manager local transaction (RMLT) which has the duration of the local transaction containment boundary (e.g. the remotable method boundary). The container commits each RMLT at the end of the local transaction containment (LTC) boundary, unless an unhandled exception occurs, in which case the container aborts each RMLT at the LTC boundary.
- application – The container does not wrap interactions with resource managers in a RMLT. The application manages the start and end of its own RMLTs or gets autocommit (which commits after each use of a resource) behavior in the absence of any explicit LocalTransaction demarcation. The application MUST complete any RMLTs it starts before the end of the LTC boundary otherwise the container will clean up such *dangling RMLTs* by aborting them.

OneWay method Implementations

OneWay method implementations may specify a transaction environment, but there are some additional semantics to consider.

When the component's transport bindings support transacted receipt and delivery semantics, and the component is running within a global transaction, the SCA runtime will ensure that the receipt of input messages and delivery of asynchronous output messages are committed/rolled-back based on the outcome of the enclosing global transaction.

When running in a "local" transaction, the SCA runtime will ensure that receipt of input messages and sending of callback response messages occurs immediately. If a runtime exception occurs during execution of the business logic, the input message is not re-delivered and previously delivered output messages will still have been delivered.

OneWay method implementations which run under "any" transaction will experience the same semantics as "local" due to a restriction that OneWay implementations cannot join a propagated global transaction. See the joinsTransaction policy below for more details.

3.6.2.2 joinsTransaction

This declaration is associated with a service interface and describes how a propagated transaction context should be handled by the SCA runtime, prior to dispatching a service component. Propagated transactions can be joined or ignored. If the client is running within a transaction that is joined by a service provider, then the primary business effects of the provider's operation are covered by the client's transaction – if the client rolls back its transaction, then work associated with the provider's operation will also be rolled back. This allows clients to know that no compensation is necessary when rollback can be used.

This declaration specifies a contract that MUST be implemented by the SCA container, but which does not impose on or restrict a client's transactional environment. It merely exposes the contractual obligation that the target component is providing. This aspect of a service component is most likely captured during application design. The valid values are:

- **true** (default) – The hosting container will join any propagated (client) transaction.
- **false** – The hosting container will not join any propagated (client) transaction.

The SCA runtime ignores joinsTransaction for OneWay methods, and behaves as if joinsTransaction was set to false.

The default value of true aligns semantically with the default value of the transaction implementation policy (any).

This declaration is independent from the implementation's transaction policy and provides no information about the implementation's transaction environment.

The combination of ***joinsTransaction*** and ***transaction*** defines the following behavior for the target component:

joinsTransaction =	Transaction=	Results
True	global	Component runs in propagated transaction if present, otherwise a new global transaction
	local	Local services run in the client's local transaction. For remotable services it generates an "incompatible deployment" Error
	any	Component runs in propagated transaction if present, otherwise local transaction
False	global	Component runs in

		new global transaction
	local	Component runs in a new local transaction
	any	Component runs in a new local transaction

In the case where the **joinsTransaction** policy conflicts with the component's transaction value of *local,* an appropriate error message must be issued at deployment. SCA tooling may also detect the error earlier in the development process.

[**Note**: It may also be valuable to introduce a declaration that a client can place on references that would be a way for the client to require that transactions be propagated (possibly called ***mustJoinTransaction***). This would be a way for the client developer to tell the deployer that they are depending on the transaction being propagated, perhaps because no compensation code was written. This means that the reference must be wired to a service that has ***joinsTransaction***=True and the binding must be able to propagate the transaction.]

3.6.2.3 suspendClientTransaction

SCA assumes that transaction context is propagated by default to target services when the target service is invoked using the synchronous SCA API, and when the bound transport and protocol support a transaction context. SCA recognizes use cases where clients wish to invoke services outside the scope of any client transaction. SCA allows a reference to be annotated with the ***suspendClientTransaction*** declaration, which prevents propagation of transaction context. The following are valid values:

- **false** - (default) Invocations of the target service run within any client global transaction. In the presence of a client local transaction, the value of "false" produces an identical transaction semantic as the value "true" for remotable services, because local transactions are never propagated to remotable target components.
- **true** - Invocations on the reference occur outside any client global transaction. This may be achieved by the client's runtime environment suspending any global transaction context under which the client executes, for the duration of the invocation. The target service's transactional environment is not affected by this policy.

This declaration is ignored if the client invokes a OneWay method, because a client transaction is never propagated over a OneWay operation.

3.6.2.4 invokeAsync

When OneWay methods are called on a service interface, any client transaction is not propagated. However, the invocation itself could be transacted as part of any client transaction. SCA recognizes that there are use cases when invocations should occur within or outside the scope of any client transaction. SCA supports this through a reference declaration called ***invokeAsync***, which has the following values:

- **atCommit** – When running in a global transaction, OneWay invocation messages are not sent until commit has succeeded. When running in a local transaction, and the

localTranResolver attribute is set to “container” then the message is sent at the LTC boundary. If ***localTranResolver*** is set to “application” then the invocation message is sent immediately. If the binding supports transactional message sending, the message will be sent if and only if the transaction commits. If the binding does not support transactional message sending, then a deployment error occurs.

- **immediate** (default) - OneWay invocations using the reference will occur independently from the outcome of an any client transaction.

[**Note:** The Client and Implementation specification will need to specify the semantics of oneway sends. For example, can a oneway send result in a synchronous Runtime exception related to protocol error that occurs during the send?]

3.6.2.5 Transaction Example

The following example shows each of transaction polices in use.

```
<?xml version="1.0" encoding="UTF-8"?>
<sca:componentType xmlns:sca=" http://www.osoa.org/xmlns/sca/0.9"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation=" http://www.osoa.org/xmlns/sca/0.9 sca.xsd ">

   <sca:service.enterprise name="TravelService" joinsTransaction="true">
         <sca:interface />
   </sca:service.enterprise>

   <sca:reference.enterprise name="FlightService" suspendClientTransaction="true">
         <sca:interface />
   </sca:reference.enterprise>

   <sca:reference.enterprise name="CarRentalService" invokeAsync="immediate">
         <sca:interface />
   </sca:reference.enterprise>

   <sca:reference name="HotelService">
         <sca:interface />
   </sca:reference>

   <sca:implementationPolicyProfile.enterprise transaction="global" />

</sca:componentType>
```

3.6.3 Reliable Messaging

We define the following policies for security to be a base part of SCA:

Interaction Policies

- reliableDelivery

Implementation Policies

- none identified at present

3.7 Profile Types

As mentioned previously, it is commonly more useful to aggregate individual policies together under a single profile. The WS-I organization has defined the Basic Profile (BP 1.1) and Basic Security Profile (BSP 1.0) to describe base conformance profiles for Web services. The BSP further constrains the requirements of the BP. IBM has also defined a Reliable Access and Messaging Profile (RAMP 1.0) that additionally constrains these profiles with respect to reliable messaging.

To enable SCA users to simply denote that they wish to conform to these profiles, the following well-known Profile elements should be available by all SCA runtimes for entry points and external services.

- WSI_BP_1.1
- WSI_BSP_1.0
- IBM_RAMP_1.0

Additionally, a standard implementation policy profile to represent common enterprise level quality of service is defined, for use with all component types:

- implementationProfile.enterprise

They are described in more detail in the following sections. However, the profile elements represent short hand ways of using the necessary SCA policies to conform to the WS-I and IBM profiles.

3.7.1 WSI_BP_1.1

This profile provides the basic Web services interoperability profile as defined by the WS-Interoperability organization.

In the following snippet we show the schema for the SCA profile that embodies this profile.

```
<element name="WSI_BP_1.1" type="sca:WSBasicProfile"
      substitutionGroup="sca:binding"/>
<complexType name="WSBasicProfile">
      <complexContent>
            <extension base="sca:WebServiceBinding">
            </extension>
      </complexContent>
</complexType>
```

3.7.2 WSI_BSP_1.0

This profile provides the basic profile for Web services security, as defined by the WS-Interoperability organization.

In the following snippet we show the schema for the SCA profile that embodies this.

```
<element name="WSI_BSP_1.0" type="sca:WSBasicSecurityProfile"
            substitutionGroup="sca:binding"/>
<complexType name="WSBasicSecurityProfile">
      <complexContent>
            <extension base="sca:WSBasicProfile">
```

```
                    <attribute name="messageProtection"
                               type="sca:messageProtectionPolicy"
                               default="integrity"/>
                    <attribute name="authentication"
                               type="sca:authenticationPolicy"
                               default="basic"/>
              </extension>
        </complexContent>
  </complexType>

  <element name="operation.WSI_BSP_1.0"
              type="sca:WSOperationBasicSecurityProfile"
              substitutionGroup="sca:operation"/>
  <complexType name="WSOperationBasicSecurityProfile">
        <complexContent>
              <extension base="sca:WSOperationBinding">
                    <attribute name="messageProtection"
                               type="sca:messageProtectionPolicy"
                               default="integrity"/>
              </extension>
        </complexContent>
  </complexType>
```

As can be seen, it builds on the WS-I Basic Profile to further add security policies to the profile. What is more, it also consists of a sub-profile to allow explicit override of individual operations. This is to allow a Service to have individual control over the quality of service offered by operations, rather than the entire endpoint. This way it is possible, for example, to have a WS-I BP 1.1 compliant Service that supports the more stringent WS-I BSP 1.0 profile on a particular operation.

3.7.3 IBM_RAMP_1.0

This profile provides a more complete profile required for reliable business to business interactions, as currently defined by IBM's Reliable Access and Messaging Profile.

In the following snippet we show the schema for the SCA profile that embodies this profile, again, including it's operation level override.

```
  <element name="IBM_RAMP_1.0" type="sca:WSReliableAccessMessagingProfile"
                    substitutionGroup="sca:binding"/>
  <complexType name="WSReliableAccessMessagingProfile">
        <complexContent>
              <extension base="sca:WSBasicSecurityProfile">
                    <attribute name="reliableDelivery"
                               type="sca:reliableDeliveryPolicy"
                               default="true"/>
              </extension>
        </complexContent>
  </complexType>
  <element name="operation.IBM_RAMP_1.0"
              type="sca:WSOperationReliableAccessMessagingProfile"
              substitutionGroup="sca:operation"/>
  <complexType name="WSOperationReliableAccessMessagingProfile">
        <complexContent>
              <extension base="sca:WSOperationBasicSecurityProfile">
```

```
                  <attribute name="reliableDelivery"
                              type="sca:reliableDeliveryPolicy"
                              default="true"/>
            </extension>
      </complexContent>
</complexType>
```

As can been seen, it builds on the WS-I Basic Security Profile (and WS-I Basic Profile) to add reliable messaging policy.

3.7.4 implementationProfile.enterprise

```
<element name="implementationProfile.enterprise"
                  type="sca:ImplementationProfileEnterprise"
                  substitutionGroup="sca:implementationProfile" />
<complexType name="ImplementationProfileEnterprise">
      <complexContent>
            <extension base="sca:ImplementationProfile">
                  <attribute name="transaction"
                              type="sca:ImplementationTransactionPolicy"
                              default="local" />
                  <attribute name="identity"
                              type="sca:identityPolicy"
                              use="required"
                              default="system"/>
            </extension>
      </complexContent>
</complexType>

<element name="service.enterprise" type="sca:ServicePolicyProfileEnterprise"
                  substitutionGroup="sca:service" />
<complexType name="ServiceProfileEnterprise">
      <complexContent>
            <extension base="sca:ServicePolicyProfile">
                  <attribute name="joinsTransaction"
                              type="sca:joinsTransactionPolicy"
                              use="required" />
                  <attribute name="permission"
                              type="sca:permissionPolicy"
                              use="required" />
            </extension>
      </complexContent>
</complexType>

<element name="serviceOperation.enterprise"
            type="sca:ServiceOperationPolicyProfileEnterprise"
            substitutionGroup="sca:serviceOperation"/>
<complexType name="ServiceOperationProfileEnterprise">
      <complexContent>
            <extension base="sca:InterfaceOperationProfile">
                  <attribute name="joinsTransaction"
                              type="sca:joinsTransactionPolicy"
                              use="required" />
                  <attribute name="permission"
                              type="sca:permissionPolicy"
                              use="required" />
            </extension>
```

```
        </complexContent>
    </complexType>
```

As can be seen from this profile, it allows for the expression of transaction identity policy for the entire component, and for individual services the specification of transaction and permission policies.

4 References

[1] SCA Client and Implementation Specification

Any one of:

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

[2] SDO Specification

Any one of:

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

[3] SCA Example Code document

Any one of:

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

[4] JAX-WS Specification

http://jcp.org/en/jsr/detail?id=101

[5] WS-I Basic Profile

http://www.ws-i.org/deliverables/workinggroup.aspx?wg=basicprofile

[6] WS-I Basic Security Profile
http://www.ws-i.org/deliverables/workinggroup.aspx?wg=basicsecurity

[7] Business Process Execution Language (BPEL)
http://www.oasis-open.org/committees/documents.php?wg_abbrev=wsbpel

[8] WSDL Specification
WSDL 1.1: http://www.w3.org/TR/wsdl
WSDL 2.0: http://www.w3.org/TR/wsdl20/